UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

EVALUACIÓN DE SPRING MVC

David Mayor Martín

Septiembre / 2014

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

EVALUACIÓN DE SPRING MVC

Autor: **David Mayor Martín**

Director: **Salvador Otón Tortosa**

Tribunal:

Presidente: _____

Vocal 1º: _____

Vocal 2º: _____

Calificación: _____

Alcalá de Henares a de de 2014

ÍNDICE RESUMIDO

ÍNDICE DETALLADO

ÍNDICE DE FIGURAS

OBJETIVO DEL TRABAJO

Spring Framework cuenta con su propio framework de aplicaciones web basado en el Modelo Vista Controlador (MVC). Los desarrolladores de Spring decidieron escribir su propio framework web como una reacción a lo que percibían como el mal diseño de Jakarta Struts framework web, así como por las deficiencias en otros frameworks disponibles. En particular, sentían que no había la separación suficiente entre las capas de presentación y tratamiento de la petición, y entre las capas de tratamiento de la petición y el modelo.

Al igual que Struts, Spring MVC es un framework basado en peticiones, además define el patrón Strategy que da a las interfaces todas las responsabilidades que debería tener un framework moderno basado en peticiones. El objetivo de cada interfaz debe ser simple y claro para que sea más fácil para los usuarios de Spring MVC escribir sus propias implementaciones. MVC allana el camino para conseguir un código limpio y reutilizable. Todas las interfaces están estrechamente unidas a la API Servlet, con esta relación se asegura que las características de la API de Servlet sigan estando disponibles para los desarrolladores, al tiempo que se ofrece un alto nivel de abstracción para facilitar el trabajo con dicha API.

El controlador principal de la estructura y el responsable de la delegación del control a las diversas interfaces durante las fases de ejecución de una solicitud HTTP es la clase DispatcherServlet.

Con este proyecto se busca realizar una evaluación del framework Spring MVC para determinar su arquitectura, características y sus ventajas e inconvenientes en relación a otros frameworks. Para ello se realizara un estudio del modelo vista controlador en el que se basa el framework y del framework en sí mismo para determinar sus características con respecto a otros frameworks. Por último se realizara una aplicación basada en Spring MVC en las que se pondrá en práctica todos los conocimientos aprendidos en los estudios anteriores.

CONCEPTOS BÁSICOS

1. Aplicaciones Web

Se denomina **aplicación web** a cualquier aplicación que los usuarios pueden utilizar accediendo a un servidor web a través de Internet o de una Intranet.

En general, el término también se utiliza para designar aquellos programas informáticos que son ejecutados en el entorno del navegador o codificado con algún lenguaje soportado por el navegador (como JavaScript, HTML, PHP, Perl, etc.); confiándo en el navegador web para que reproduzca la aplicación.

El uso de frameworks en la construcción y diseño de las aplicaciones Web reducen el tiempo de desarrollo, permitiendo mejorar la productividad de los sistemas, además de proporcionar un diseño robusto y extensible.

2. Tecnologías Java usadas en aplicaciones Web

Las tecnologías sobre las que se apoyan las aplicaciones Web en Java son:

- Protocolo HTTP.
- API Java Servlet.
- Java Server Pages (JSP).

Tanto la tecnología Servlet como la JSP están íntimamente relacionadas, ya que las paginas JSP al final se convierten en Servlets que son las aplicaciones que realmente se ejecutan en el servidor web.

Para simplificar la complejidad de los desarrollos e intentar aliviar el exceso de carga asociado a las actividades de las aplicaciones Web, surgen componentes de programación que permiten abstraer la construcción de aplicaciones Web. La unión de los componentes recibe el nombre de Framework Web.

2.1. Hypertext Transfer Protocol (HTTP)

El protocolo HTTP es un sencillo protocolo cliente-servidor, que permite la transferencia de archivos (principalmente, en formato HTML) entre un navegador (el cliente) y un servidor web (el servidor) localizado mediante una cadena de caracteres denominada dirección URL. Una **transacción HTTP** se compone de una cabecera en la que se especifica tanto la acción solicitada en el servidor como los tipos de datos devueltos, o un código de estado.

Entre las propiedades del protocolo HTTP se pueden destacar:

- Arquitectura Cliente-Servidor: HTTP se asienta en el paradigma solicitud/respuesta. La comunicación se asienta sobre TCP/IP.
- Esquema de direccionamiento comprensible: Utiliza el Universal Resource Identifer (URI) para localizar sitios (URL) o nombres (URN) sobre los que aplicar un método.

- HTTP es un protocolo sin estado, es decir, que no guarda ninguna información sobre conexiones anteriores.
- Está abierto a nuevos tipos de datos: HTTP utiliza tipos **MIME (*Multipurpose Internet Mail Extensions*)** para la determinación de tipos de datos que transporta.

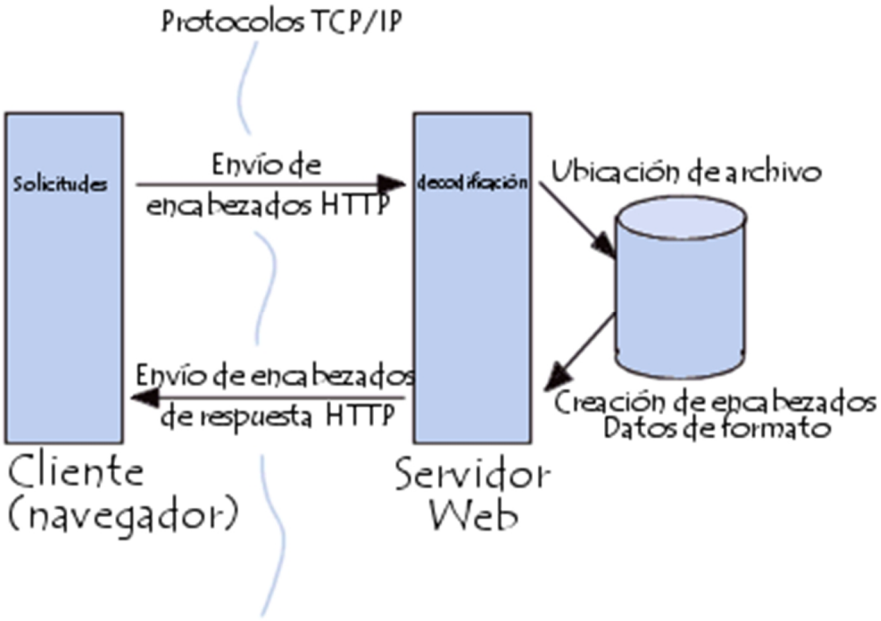


Figura 1. Esquema del protocolo HTTP

2.2. API Java Servlet

La **Api Java Servlet** ayuda a solventar algunos de los problemas que nos plantea el **protocolo HTTP**, realizando una abstracción de las peticiones y las respuestas, para poder trabajar con objetos Java permitiendo a los desarrolladores manejar las respuestas HTTP.

La figura central de **API Java Servlet** son los **Servlets.** Se puede definir un **Servlet** como un **programa JAVA** que se ejecuta en un entorno distribuido de red, como **un servidor web**, y que recibe y responde las peticiones de un cliente a través del protocolo **HTTP**.

Para la utilización de los Servlets se deben en tener cuenta los siguientes aspectos:

- Son independientes del servidor y de la plataforma.
- Se suelen utilizar en aplicaciones donde sea necesario el manejo de los datos que incorpore el usuario, además del acceso de Bases de Datos remotas y el manejo de sesiones.
- El servidor debe de disponer de una máquina virtual Java, además debe soportar la API de los Servlets.

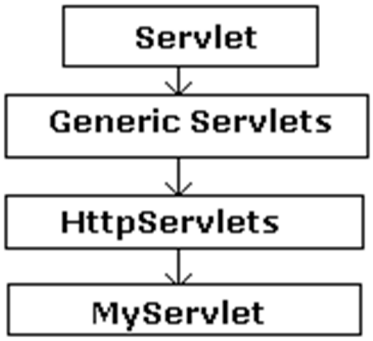


Figura 2. Dependencias de las clases de Servlets

Otras de las partes fundamentales de los Servlets son los objetos de petición (**HttpServletRequest**), que se encargan de encapsular las funcionalidades de las peticiones del cliente, y los objetos de respuesta (**HttpServletResponse**), que se encargan de encapsular la funcionalidad de una respuesta HTTP, incluyendo el manejo de las cabeceras del propio protocolo.

Para comprender mejor el funcionamiento de un Servlet se procederá a explicar su ciclo de vida:

1.-Un servidor carga e inicializa el Servlet.

2.-El Servlet maneja cero o más peticiones de cliente.

3.-El servidor elimina el Servlet.

1.-Inicializar un Servlet

Cuando un servidor carga un Servlet, ejecuta el método init del Servlet. La inicialización se completa antes de manejar peticiones de clientes y antes de que el Servlet sea destruido.

Aunque muchos Servlets se ejecutan en servidores multi-hilos, los Servlets no tienen problemas de concurrencia durante su inicialización. El servidor llama sólo una vez al método init al crear la instancia del Servlet.

2.-Interactuar con Clientes

Después de la inicialización, el Servlet puede manejar peticiones de clientes. Estas respuestas son manejadas por la misma instancia del Servlet.

3.-Destruir un Servlet

Los Servlets se ejecutan hasta que el servidor los destruye, por el cierre del servidor o bien a petición del administrador del sistema. Cuando un servidor destruye un Servlet, ejecuta el método destroy del propio Servlet. El servidor no ejecutará de nuevo el Servlet, hasta haberlo cargado e inicializado de nuevo.

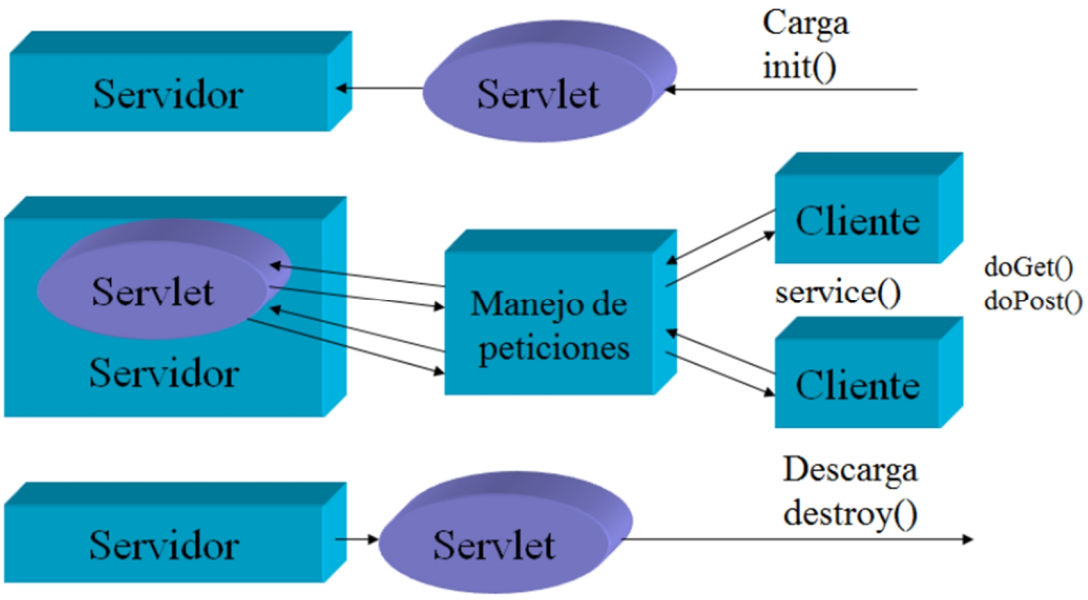


Figura 3. Ciclo de vida de un Servlet

2.3. Java Server Pages

Java Server Pages (JSP) es una tecnología que ayuda a los desarrolladores de software a crear páginas web dinámicas basadas en HTML, XML entre otros tipos de documentos. El contenido dinámico se obtiene, en esencia, gracias a la posibilidad de incrustar dentro de la página código Java de distintas formas.

Las paginas **JSP** se convierten en un **Servlet**, esta conversión es realizada por la máquina servidora mediante el motor JSP la primera vez que se solicita la página al servidor web. Este Servlet generado se almacena en caché para ser reusado hasta que se modifique la página JSP original, cuando esto sucede automáticamente el Servlet se regenera y recompila para recargarse la próxima vez que sea solicitado. El Servlet generado a partir del JSP procesa cualquier petición para esa página JSP.

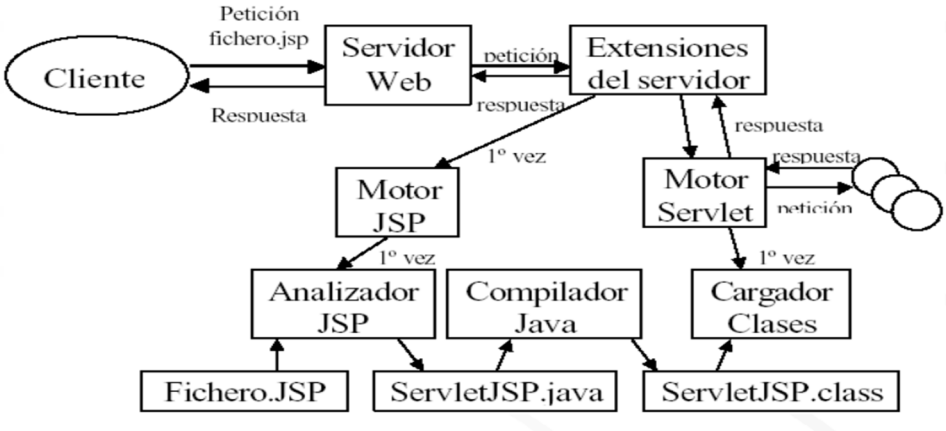


Figura 4.Ciclo de vida de un jsp

El rendimiento de una página **JSP** es el mismo que tendría el Servlet equivalente, ya que el código es compilado como cualquier otra clase Java. A su vez, la máquina virtual compilará dinámicamente a código de máquina las partes de la aplicación que lo requieran. Esto hace que JSP tenga un buen desempeño y sea más eficiente que otras tecnologías web que ejecutan el código de una manera puramente interpretada. El ciclo de vida de las paginas JSP es exactamente igual al de los Servlets pero llamando a los métodos propios de los JSP.

La principal ventaja de **JSP** frente a otros lenguajes es que el lenguaje Java es un lenguaje de propósito general que excede el mundo web y que es apto para crear clases que manejen lógica de negocio y acceso a datos de una manera prolija. Esto permite separar en niveles las aplicaciones web, dejando la parte encargada de generar el documento HTML en el archivo JSP.

2.4. Frameworks Web de Java

Una de las herramientas que surgen para simplificar la tarea del desarrollo software y fomentar la modularidad, son los frameworks o marcos de trabajo.

Un framework se define como aquella aplicación o conjunto de módulos que permiten, o tienen por objetivo, el desarrollo ágil de aplicaciones mediante la aportación de librerías y/o funcionalidades ya creadas para que nosotros las usemos directamente. Un framework tiene las siguientes características:

- Un framework consta de múltiples clases o componentes, cada uno de los cuales puede proveer una abstracción de un determinado concepto.
- El framework define como esas abstracciones trabajan juntas para solucionar el problema.
- Los componentes del framework son reutilizables.
- Un framework organiza patrones a alto nivel.

Un buen framework debería proveer de un comportamiento genérico para que distintos tipos de aplicaciones puedan hacer uso de él, de manera que los programadores puedan centrarse en la implementación de las funcionalidades de la aplicación.

Las principales ventajas de la utilización de frameworks son:

- Desarrollo más rápido de las aplicaciones. Los componentes incluidos en un framework constituyen una capa que libera al programador de la estructura de código de bajo nivel.
- Mayor reutilización de componentes software.
- El uso y la programación de componentes siguen una política de diseño uniforme. Un framework orientado a objetos logra que los componentes sean clases que pertenezcan a una gran jerarquía de clases, lo que facilita el aprendizaje de la utilización de las librerías.
- El código de los frameworks está por lo general optimizado y testeado. En general un framework que cuenta con una comunidad importante siempre se encuentra optimizado. Lo mismo ocurre con aspectos fundamentales como la seguridad.

Las desventajas de la utilización de frameworks son:

- La dependencia del código fuente de una aplicación con respecto al framework. Si se desea cambiar el framework la mayor parte del código debe reescribirse.
- La demanda de grandes cantidades de recursos computacionales debido a que la característica de reutilización de los frameworks tiende a generalizar la funcionalidad de los componentes. El resultado es que el framework introduce características que están de más en determinadas aplicaciones, provocando una sobrecarga de recursos.
- Alto tiempo de aprendizaje. Aprender a utilizar un framework no es algo que se pueda hacer en un par de días. Si bien ahorramos mucho tiempo en el desarrollo de aplicaciones una vez que los programadores conoces el framework, también se tienen que dedicar muchísimas horas en aprender todos los aspectos.

No es necesario la utilización de frameworks para el desarrollo de aplicaciones Web, pero su utilización ayuda mucho a la productividad y mejora el desarrollo.

3. Historia

A continuación se mostrará la evolución de las tecnologías Web en la historia hasta llegar al nacimiento de Spring MVC.

3.1. CGI

Cuando el World Wide Web empezó a tomar popularidad como tecnología para gestionar la información, aproximadamente en 1993, el **CGI** (**Common Gateway Interface**) cambió la forma de manipular información en el web.

CGI es una importante tecnología de la World Wide Web que permite a un cliente solicitar datos de un programa ejecutado en un servidor web. CGI especifica un estándar para transferir datos entre el cliente y el programa. Es un mecanismo de comunicación entre el servidor web y una aplicación externa cuyo resultado final de la ejecución son objetos MIME. Las aplicaciones que se ejecutan en el servidor reciben el nombre de **CGIs**.

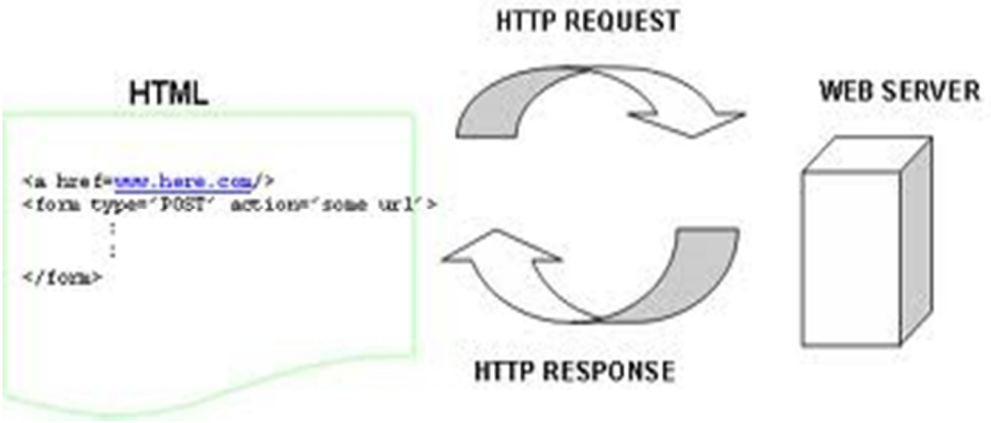


Figura 5. Esquema de funcionamiento de las peticiones web

Las aplicaciones CGI fueron una de las primeras prácticas de crear contenido dinámico para las páginas web. En una aplicación CGI, el servidor web pasa las solicitudes del cliente a un programa externo. Este programa puede estar escrito en cualquier lenguaje que soporte el servidor. La salida de dicho programa es enviada al cliente en lugar del archivo estático tradicional.

3.2. Servlet

La tecnología Servlet fue desarrollada por Sun Microsystems en 1995, siendo aprobada la versión 1.0 en junio de 1997. Esta tecnología fue considerada superior al CGI ya que los Servlets residían en memoria tras atender la primera petición HTTP, mejorando el tiempo de respuesta entre peticiones.

Histórico de las versiones API Servlet:

Versión Servlet API	Realización	Plataforma
Servlet 3.1	Mayo 2013	JavaEE 7
Servlet 3.0	Diciembre 2009	JavaEE 6, JavaSE 6
Servlet 2.5	Septiembre 2005	JavaEE 5, JavaSE 5
Servlet 2.4	Noviembre 2003	J2EE 1.4, J2SE 1.3
Servlet 2.3	Agosto 2001	J2EE 1.3, J2SE 1.2
Servlet 2.2	Agosto 1999	J2EE 1.2, J2SE 1.2
Servlet 2.1	Noviembre 1998	
Servlet 2.0		JDK 1.1
Servlet 1.0	Junio 1997	

Figura 6. Historico de versiones de Serlvlets

Los problemas con los Servlets es que resulta muy complicado y el código necesario para diseñar páginas HTML resulta complejo. Todo el código HTML se debía crear en cadenas de texto (String) para después enviarlas a través del PrintWritter del objeto HttpServletResponse.

```
PrintWritter out=responsePrintWritter;
out.println (“<html><head><title>Mi  Servlet</title></head>”);
out.println (“<body>Mi primer Servlet</body></html>”);
```

La codificación del código HTML incrustado dentro de un Servlet provoca que cada cambio que se produzca en la vista haya que recompilar la clase completa y volverla a desplegar.

3.3. JSP

Para solventar los inconvenientes que proponían los Servlets, Sun Microsystem lanzo en Junio de 1998 la tecnología JSP en la versión 0.91, ya que esta tecnología no necesitaba recompilarse para mostrar los cambios efectuados.

Pese a poderse integrar código Java en los JSP mediante scriptlets, esto provoca que el código de la vista pueda volverse muy complejo, por esta razón se empezaron a crear arquitecturas para separar la vista del controlador.

Con la versión de JSP 1.0 se permitió la inclusión de JavaBeans en el código, así como tags para trabajar con los objetos para evitar en la medida de lo posible incluir en la vista código Java. Así en la versión 1.1 se incluyeron las librerías de etiquetas que proporcionaban más flexibilidad que las directivas Java Beans.

Posteriormente se crearon las **Java Server Pages Stand Tag Libraries** (**JSTL**) que suponían un conjunto de librerías con funcionalidad común a la mayoría se desarrollos. Finalmente en la versión 2.0 se permitió a los desarrolladores deshabilitar las porciones de código Java en los JSP, además se simplifico el clico de vida de los tags.

3.4. Spring

Los primeros componentes de lo que se ha convertido en Spring Framework fueron escritos por Rod Johnson en el año 2000. Mientras escribía el libro Expert One-on-one J2EE Design And Development (Programmer to programmer), Rod amplió su código para sintetizar su visión acerca de cómo las aplicaciones que trabajan con varias partes de la plataforma J2EE podían llegar a ser más simples y más consistentes que aquellas que los desarrolladores y compañías estaban usando por aquel entonces.

En febrero de 2003 se creó un proyecto en Sourceforge en que se formó un pequeño equipo de desarrolladores que esperaba trabajar en extender el framework. Después de trabajar en su desarrollo durante más de un año lanzaron una primera versión (1.0) en marzo de 2004. Después de este lanzamiento Spring ganó mucha popularidad en la comunidad Java, debido en parte al uso de Javadoc y de una documentación de referencia por encima del promedio de un proyecto de código abierto.

Sin embargo, Spring Framework también fue duramente criticado en. Al tiempo en que se producía su primer gran lanzamiento muchos desarrolladores y líderes de opinión vieron a

Spring como un gran paso con respecto al modelo de programación tradicional. Una de las metas de diseño de Spring Framework es su facilidad de integración con los estándares J2EE y herramientas comerciales existentes. Esto quita en parte la necesidad de definir sus características en un documento de especificación elaborado por un comité oficial y que podría ser criticado.

Spring Framework hizo que aquellas técnicas que resultaban desconocidas para la mayoría de programadores se volvieran populares en un periodo muy corto de tiempo. El ejemplo más notable es la inversión de control. En el año 2004, Spring disfrutó de unas altísimas tasas de adopción y al ofrecer su propio framework de programación orientada a aspectos.

En 2005 Spring superó las tasas de adopción del año anterior como resultado de nuevos lanzamientos y más características fueron añadidas. El foro de la comunidad formada alrededor de Spring Framework (The Spring Forum) que arrancó a finales de 2004 también ayudó a incrementar la popularidad del framework y desde entonces ha crecido hasta llegar a ser la más importante fuente de información y ayuda para sus usuarios.

4. Modelos de diseño

En el diseño de aplicaciones Web Java, hay dos modelos de diseño de uso común, se conocen como el Model 1 y Model 2.

4.1. Model 1

La arquitectura de Model 1 es un acercamiento donde el centro de la lógica reside en la página que controla los flujos de página. Esto significa que el procesado de la petición y de la respuesta está incrustado directamente en la página.

En este modelo se hace una petición a un JSP o Servlet para que luego este maneje todas las responsabilidades de la solicitud, incluyendo la tramitación de la solicitud, validación de los datos, el manejo de la lógica de negocio, y la generación de la respuesta.

Evidentemente, esta arquitectura presenta notables problemas de mantenimiento cuando se producen modificaciones en la lógica necesarias para adecuar las demandas de los usuarios con nuevos requerimientos y funcionalidades. Estos cambios obligan a los desarrolladores a "peinar" el código entre las páginas para localizar donde realizar los cambios y comprobar de nuevo los flujos de navegación para asegurar un comportamiento adecuado.

Solo por esta razón sería suficiente para que su uso fuera muy limitado, sin embargo en proyectos con un equipo cuyos miembros cuenten con una muy reducida experiencia, o bien el alcance y la magnitud del proyecto sean muy limitados, o el tiempo de desarrollo hasta la entrega muy corto, para cualquier otra situación los problemas que ocasiona el tomar esta arquitectura nos debería persuadir para buscar mejores soluciones.

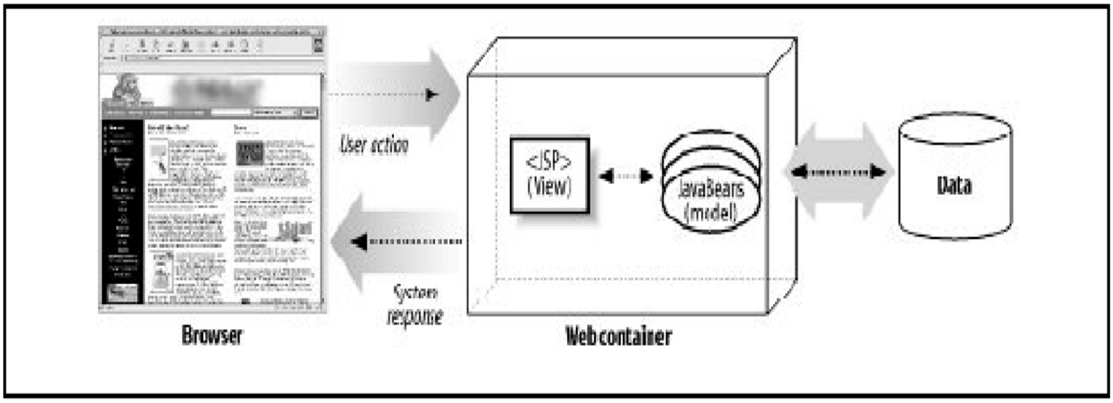


Figura 7. Esquema del Model 1

4.2. Model 2

Al contrario como hemos visto en el caso anterior, los flujos de navegación están regidos por un Servlet controlador que funciona en conjunción con ficheros de configuración para dictar como y cuando se presentan las páginas durante los operaciones de la aplicación.

El Model 2 es flexible ya que separa tu aplicación en diferentes componentes clasificados por lo que saben hacer. Esto permite insertar tantas acciones o vistas como sean necesarias sin la necesidad de tener que reescribir todo.

Otro punto fuerte es la escalabilidad, ya que al tener componentes separados, es sencillo añadir más componentes allí donde sea necesario. Además se puede crear cachés de los componentes de datos más fácilmente debido a la separación de funcionalidades. Ahora las vistas no se preocupan por si los datos que muestran provienen de datos reales o de una versión con datos cacheados.

Otro de sus puntos fuertes es la seguridad, ya que al manejar todas las acciones a través de un controlador central, se puede configurar y manejar fácilmente como controlar el acceso a los datos y a las acciones.

Sin embargo este modelo también cuenta con limitaciones y problemas podemos destacar las siguientes:

La curva de aprendizajes es importante ya que no se puede usar el Modelo 2 sino se entiende en que consiste.

Además muchos desarrolladores están acostumbrados a desarrollar sus aplicaciones de forma interactiva. Por lo que el concepto de compilación y dependencia es totalmente ajeno a ellos. Sin embargo, de nuevo los beneficios vuelven a relucir ya que al contar con los distintos componentes se puede realizar una separación de papeles entre los miembros del equipo, para colocarlos donde se ajustes sus capacidades y conocimientos.

Por lo tanto a menos que se nos pida desarrollar una web con menos de media docena de páginas siempre va a ser más rentable decantarse por el Model 2 ya que sus beneficios van a salir a relucir.

En conclusión podemos decir que en el Model 1 los beneficios se ven en un primer momento pero que a medida que cambian los requerimientos y tenemos que hacer un mantenimiento esos beneficios se ven sensiblemente reducidos hasta poder llegar a convertirse en un quebradero de cabeza. En el Model 2 sin embargo el coste de desarrollo se incrementa notablemente lo que en un principio se convierte en un obstáculo pero a medida que se incrementen funcionalidades y hay que realizar un mantenimiento los beneficios salen a flote, y no hay que olvidar que el tiempo de mantenimiento y de posteriores mejoras es varias veces superior al tiempo que se emplea normalmente en desarrollo por lo que la diferencia en ahorro de coste acaba siendo abismal.

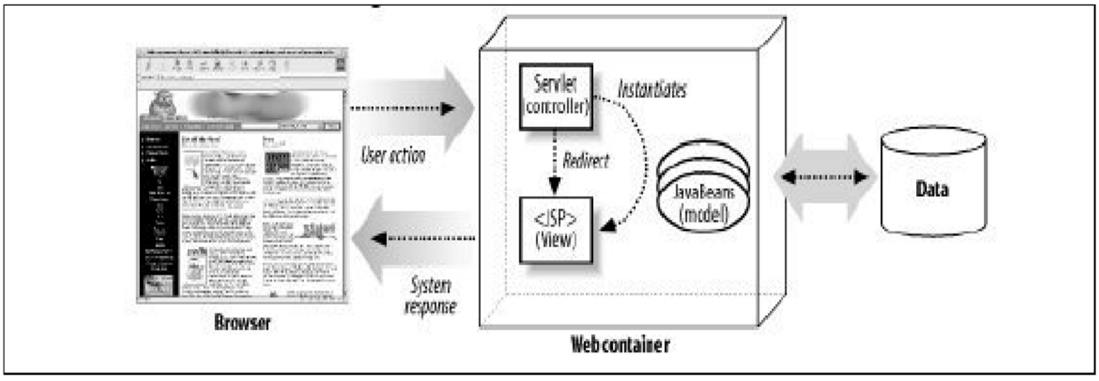


Figura 8. Esquema del Model 2

5. Patrón Modelo-Vista-Controlador

5.1. ¿Qué es un patrón?

Un patrón es una solución a un problema básico, es una información que captura la estructura esencial y la perspicacia de una familia de soluciones probadas con éxito para un problema repetitivo que surge en un cierto contexto y sistema.

La forma de obtener esta solución es a partir de la abstracción de ejemplos específicos de diseño. Para que una solución pueda ser considerada un patrón de diseño debe ser eficaz (que se haya demostrado resuelve satisfactoriamente el problema) y reutilizable (que pueda ser aplicada en diferentes casos).

Pero los patrones no se quedan simplemente ahí. Proporcionan un vocabulario común con otros desarrolladores. Una vez que se tenga el vocabulario uno puede comunicarse más eficientemente con otros desarrolladores, esto facilita el mantenimiento y las posteriores mejoras de las aplicaciones.

Si hablas de patrones se conoce de forma inmediata y precisa el diseño que estas escribiendo. También permite estructurar un nivel arquitectónico interponiendo una capa de patrones a tu pensamiento.

5.2. Modelo-Vista-Controlador (MVC)

El patrón MVC se cataloga como un patrón de diseño que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

En la sección en la que se describía el modelo 2, se mostraba la preocupación por separar las distintas responsabilidades en las aplicaciones web. Permitir una página JSP que maneje las peticiones, ejecutando la lógica de negocio y determinando las vistas a mostrarse crea un monstruo muy difícil de manejar, ya que el mantenimiento y posterior extensión hacen aflorar las limitaciones del modelo 1. El desarrollo de aplicaciones y su mantenimiento es mucho más sencillo si coexisten diferentes componentes en la aplicación web con claras y distintas responsabilidades.

El patrón MVC tiene tres componentes fundamentales:

1. **Modelo.** El Modelo es el objeto que representa los datos del programa. Maneja los datos y controla todas sus transformaciones. El Modelo no tiene conocimiento específico de los Controladores o de las Vistas, ni siquiera contiene referencias a ellos. Es el propio sistema el que tiene encomendada la responsabilidad de mantener enlaces entre el Modelo y sus Vistas, y notificar a las Vistas cuando cambia el Modelo.
2. **Vista.** La Vista es el objeto que maneja la presentación visual de los datos representados por el Modelo. Genera una representación visual del Modelo y muestra los datos al usuario. Interactúa con el Modelo a través de una referencia al propio Modelo.
3. **Controlador.** El Controlador es el objeto que proporciona significado a las órdenes del usuario, actuando sobre los datos representados por el Modelo. Cuando se realiza algún cambio, entra en acción, bien sea por cambios en la información del Modelo o por alteraciones de la Vista. Interactúa con el Modelo a través de una referencia al propio Modelo.

Iteración de los componentes.

Aunque se pueden encontrar diferentes implementaciones de **MVC**, el flujo de control que sigue generalmente es el siguiente:

1. El usuario interactúa con la interfaz de usuario.
2. El controlador recibe (por parte de los objetos de vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo. El modelo no debe tener

conocimiento directo sobre la vista. Sin embargo, los tres componentes se unen mediante un patrón Observer, que tiene como misión informar cuando un objeto cambia de estado, todas sus dependencias sean notificadas y actualizadas. En general el controlador no pasa objetos del modelo a la vista aunque puede dar la orden a la vista para que se actualice.

5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

MVC en la web.

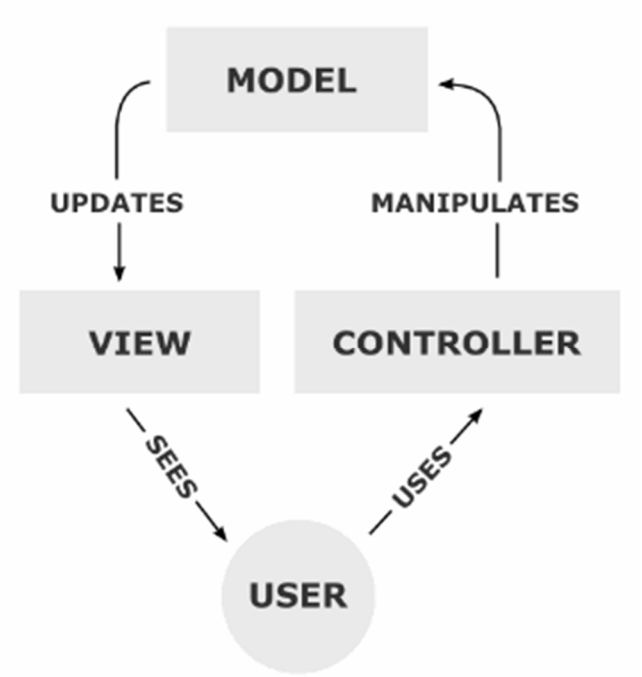


Figura 9. Funcionamiento del MVC.

En un patrón MVC, un evento notifica a la vista cuando una porción del modelo cambia. Sin embargo como un navegador en una aplicación web tiene una conexión sin estado la notificación desde el modelo a la vista no es sencilla. Aunque la aplicación permita algún mecanismo de tecnología "push" (los datos son servidos sin previa petición) normalmente llevar esto a cabo destroza la arquitectura web.

En las aplicaciones Web, un cliente envía una petición al servidor para conocer si se han producido cambios en el modelo. Esto es lo que se conoce por acercamiento "pull".

Una de las razones por las que le modelo 2 nunca logrará ser un MVC puro es que el patrón Observer que forma parte del MVC no funciona bien para un entorno web. Como se ha comentado anteriormente HTTP es un protocolo "pull": el cliente envía peticiones y el servidor devuelve respuestas. Sin petición no hay respuesta. El patrón Observer requiere un protocolo "push" para la notificación, así el servidor puede enviar un mensaje al cliente cuando el modelo cambie. Aunque hay formas de simular el flujo de datos "push" a un cliente web, no deja de ser una triquiñuela.

El Modelo en la web.

Dependiendo del tipo de arquitectura que emplee la aplicación, el modelo puede tomar diferentes formas. En una aplicación de dos capas, donde la capa web interacciona directamente con un almacén de datos, las clases del modelo pueden ser un conjunto de clases estándar de java.

En una aplicación empresarial más compleja, el modelo estará formado por los EJBs (Enterprise JavaBeans).

La Vista en la web.

Las vistas dentro de la capa web típicamente consisten en HTML y páginas JSP que proporcionan contenido estático o dinámico respectivamente.

La mayoría del contenido dinámico es generado en la capa web. Aunque algunas aplicaciones requieren que ese contenido dinámico se realice en el lado del cliente.

Además HTML y JSP no son las únicas opciones para las vistas. Se puede dar soporte WML (WAP) para móviles, o se pueden utilizar otros motores de renderizado. Dado que la vista no está ligada al modelo, se permite soportar distintas vistas para cada cliente, usando el mismo componente modelo.

El Controlador en la web.

El controlador en la web suele estar diseñado para un Servlet, sus deberes consisten en:

1. Interceptar las peticiones HTTP del cliente.

2. Traducir cada petición en una operación específica de negocio a ser realizada.

3. Invocar o bien delegar en un manejador la operación.

4. Ayudar a seleccionar la siguiente vista a mostrar al cliente.

5. Devolver la vista al cliente.

Existe un patrón que define como se debe implementar un controlador, es el Front Controller que forma parte de la plataforma J2EE de patrones de diseño. Ya que todas las peticiones y respuestas pasan a través del controller, existe un punto centralizado de control en la aplicación web. Esto ayuda cuando hay que añadir nueva funcionalidad. El controlador también ayuda a desligar los componentes de presentación (vistas) de las operaciones de negocio.

INTRODUCCIÓN A SPRING FRAMEWORK

Spring Framework es una plataforma Java que proporciona un amplio soporte de infraestructuras para el desarrollo de aplicaciones Java. Se encarga de la infraestructura para que el desarrollador pueda centrarse en su aplicación.

Spring permite construir aplicaciones POJOs (un objeto POJO es una instancia de una clase que no extiende ni implementa nada en especial) y solicitar los servicios de la aplicación de forma no invasiva para los POJOs. Esta capacidad está relacionada con el modelo de programación de Java SE y Java EE.

Ejemplos de cómo un desarrollador de aplicaciones, puede utilizar las ventajas de Spring:

- Crea un método Java para ejecutar una transacción de base de datos sin tener que lidiar con las APIs de transacción.
- Crea un método local Java como un procedimiento remoto sin tener que lidiar con las APIs remotas.
- Crea un método local Java como una operación de gestión sin tener que lidiar con las APIs de JMX.
- Crea un método local Java como un controlador de mensajes sin tener que lidiar con las APIs de JMS.

1. Inyección de Dependencia e Inversión de Control

Las aplicaciones Java (término que abarca desde componentes restringidos hasta aplicaciones empresariales) suelen consistir en objetos que colaboran para formar una aplicación. Así, los objetos de una aplicación, tienen dependencias entre sí.

Aunque la plataforma Java proporciona una gran cantidad de funcionalidades en el desarrollo de aplicaciones, estas carecen de medios para organizar los bloques básicos de construcción en un todo coherente, dejando esta tarea a los diseñadores y desarrolladores del sistema. Es cierto que se pueden utilizar patrones de diseño, tales como Factory Method, Abstract Factory, Builder, Decorator, and Service Locator para crear las distintas clases e instancias de objetos que forman la aplicación. Sin embargo, estos patrones son simplemente las mejores prácticas que se deben aplicar a la hora de desarrollar una aplicación. Los patrones se componen de la descripción de lo que hace el patrón, dónde aplicarlo, los problemas que aborda y como solucionar dichos problemas.

La inversión de control (IoC) de Spring Framework es un componente que se encarga de proporcionar un medio formalizado para crear los diversos componentes de una aplicación funcional lista para su uso inmediato. Spring Framework codifica patrones de diseño formalizados como objetos de primera clase que se puede integrar en la propia aplicación. Numerosas organizaciones e instituciones utilizan Spring Framework de esta manera para diseñar aplicaciones.

2. Módulos

Spring Framework consta de funciones organizadas en cerca de 20 módulos. Estos módulos se agrupan en Core Container, Data Access / Integration, Web, AOP (Programación orientada a Aspectos), Instrumentation y Test, y están distribuidos tal y como se muestra en el siguiente diagrama.

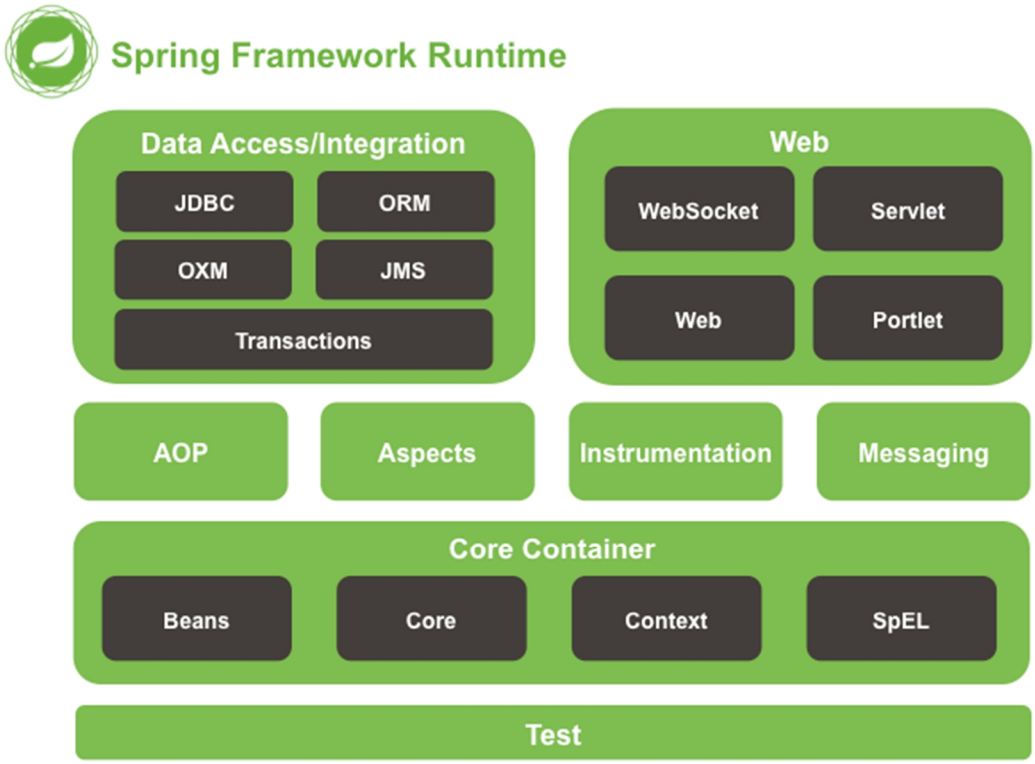


Figura 10. Estructura de Spring Framework.

2.1. Core Container

El **Core Container** está formado por: Core, Beans, Context y SpEL.

Core y Beans son módulos que proporcionan las partes fundamentales de la estructura, incluyendo las características de la IoC y de la inyección dependencia. El BeanFactory es una sofisticada implementación del patrón Factory Method. Se elimina la necesidad de singletons y permite desacoplar la configuración y especificación de las dependencias de la lógica del programa real.

El módulo Context se centra en la sólida base proporcionada por los módulos Core y Beans: es un medio para acceder a los objetos de una manera similar a un registro JNDI. El módulo Context hereda sus características desde el módulo de Beans y añade soporte para la internacionalización (utilizando, por ejemplo, los paquetes de recursos), eventos de propagación, carga de recursos y la creación transparente de contextos (por ejemplo, un

contenedor de servlets), también es compatible con las características de Java EE como EJB, JMX, y la interacción remota básica. La interfaz ApplicationContext es el punto central del módulo Context.

El SpEL o módulo de lenguaje de expresión proporciona un poderoso lenguaje de expresión para consultar y manipular un gráfico de objetos en tiempo de ejecución. Es una extensión del lenguaje de expresiones unificado (EL), especificado en la especificación JSP 2.1. El lenguaje soporta la asignación y obtención de valores, la asignación de propiedades, la invocación de métodos, el acceso al contexto de arrays, colecciones e índices, operadores lógicos y aritméticos, nombre de variables y la recuperación de objetos por nombre del contenedor IoC de Spring. También es compatible con la lista de proyección y selección, así como la lista de las agrupaciones comunes.

2.2. Data Access / Integration

La capa Data Access / Integration está formada por: JDBC, ORM, OXM, JMS y los módulos de transacción.

El módulo JDBC proporciona una capa de abstracción JDBC que elimina la necesidad de hacer la codificación JDBC y el análisis de los códigos de error específicos de las bases de datos de los proveedores.

El módulo ORM proporciona las capas de integración para APIs de mapeo de objeto-relacional, incluyendo JPA, JDO e Hibernate. Usando el paquete ORM se pueden utilizar todos estos frameworks de mapeo de objeto-relacional en combinación con todas las otras ofertas características de Spring, como la característica de gestión de transacciones declarativas.

El módulo OXM proporciona una capa de abstracción que apoya las implementaciones de mapeo Object/XML para JAXB, Castor, XMLBeans, JiBX y xstream.

El módulo Java Messaging Service (JMS) contiene características para producir y consumir mensajes.

El módulo de transacción soporta la gestión de transacciones programáticas y declarativas para las clases que implementan interfaces especiales y para todos los POJOs.

2.3. Web

La capa Web está formado por: Web, Web-Servlet, WebSocket y los módulos en la Web de portlets.

El módulo Spring Web proporciona funciones de integración básicas de web como la funcionalidad de carga de archivos y la inicialización del contenedor IoC usando detectores de servlet y un contexto de aplicación orientado a la web. También contiene las partes relacionadas con el soporte remoto de la comunicación web de Spring.

El modulo Web-Servlet contiene la implementación del modelo-vista-controlador (Spring MVC) para aplicaciones web. Spring MVC Framework proporciona una separación limpia entre el código del modelo del dominio y las vistas de la web, integrada con todas las otras características de Spring Framework.

El módulo Web de portlets proporciona la implementación MVC para ser utilizado en un entorno de portlets y refleja la funcionalidad del módulo Web-Servlet.

2.4. AOP e Instrumentation

El módulo AOP de Spring proporciona una implementación de la programación orientada a aspectos compatible que le permite definir, por ejemplo, puntos de corte para desacoplar limpiamente el código que implementan la funcionalidad que debe ser separada. El uso de la funcionalidad de metadatos a nivel de fuente, también puede incorporar la información del comportamiento en el código, de una manera similar a la de los atributos .NET.

El módulo independiente Aspects proporciona integración con AspectJ.

El módulo Instrumentation proporciona apoyo instrumental a las clases e implementaciones classloader para ser utilizadas en ciertos servidores de aplicaciones.

2.5. Test

El módulo Test soporta las pruebas de los componentes de Spring con JUnit o TestNG. Se proporciona una carga consistente para el resorte ApplicationContexts y para el almacenamiento en caché de los contextos. También proporciona los objetos de imitación que se pueden utilizar para probar el código de forma aislada.

3. Escenarios de uso

Los bloques descritos anteriormente hacen de Spring Framework una opción lógica en muchos escenarios, como por ejemplo applets de aplicaciones que utilizan la funcionalidad de gestión de transacciones de Spring y la integración con el framework web.

Las características de gestión de transacciones declarativa de Spring hacen que la aplicación web sea completamente transaccional, tal como lo sería si se usasen transacciones gestionadas por un contenedor EJB. Toda la lógica de negocios personalizada puede implementarse con POJOs simples y es gestionada por el contenedor IoC de Spring. Los servicios adicionales incluyen soporte para el envío de correos electrónicos y la validación independiente de la capa web, que permite elegir dónde ejecutar las reglas de validación. El soporte ORM de Spring se integra con JPA, Hibernate y JDO. Los controladores de los formularios integran a la perfección la capa de web con el modelo del dominio, eliminando la necesidad de ActionForms u otras clases que transforman parámetros HTTP en valores de su modelo de

dominio. En el siguiente diagrama se muestra la organización interna de una aplicación web típica de Spring.

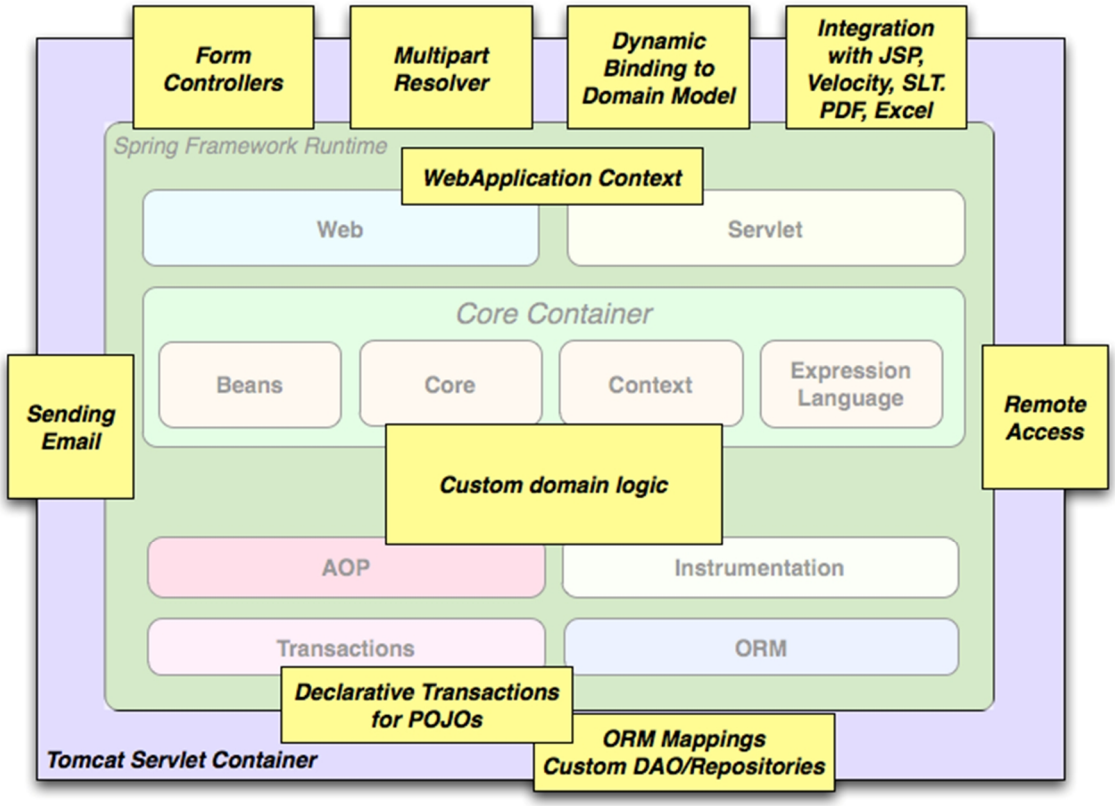


Figura 11. Diagrama de una aplicación Spring.

A veces las circunstancias no le permiten al desarrollador cambiar por completo a un marco diferente, por ello Spring Framework no obliga a usar todo los módulos que implementa. Existen interfaces construidas con Struts, Tapestry, JSF u otros frameworks de interfaz de usuario que se puede integrar con Spring, además, esto sigue permitiendo utilizar las funciones de transacción de Spring. El desarrollador sólo tendrá que conectar su lógica de negocio utilizando una ApplicationContext y utilizar un WebApplicationContext para integrar su capa web. En el siguiente diagrama se muestra la organización interna de una aplicación Spring de nivel medio utilizando un framework web de terceros.

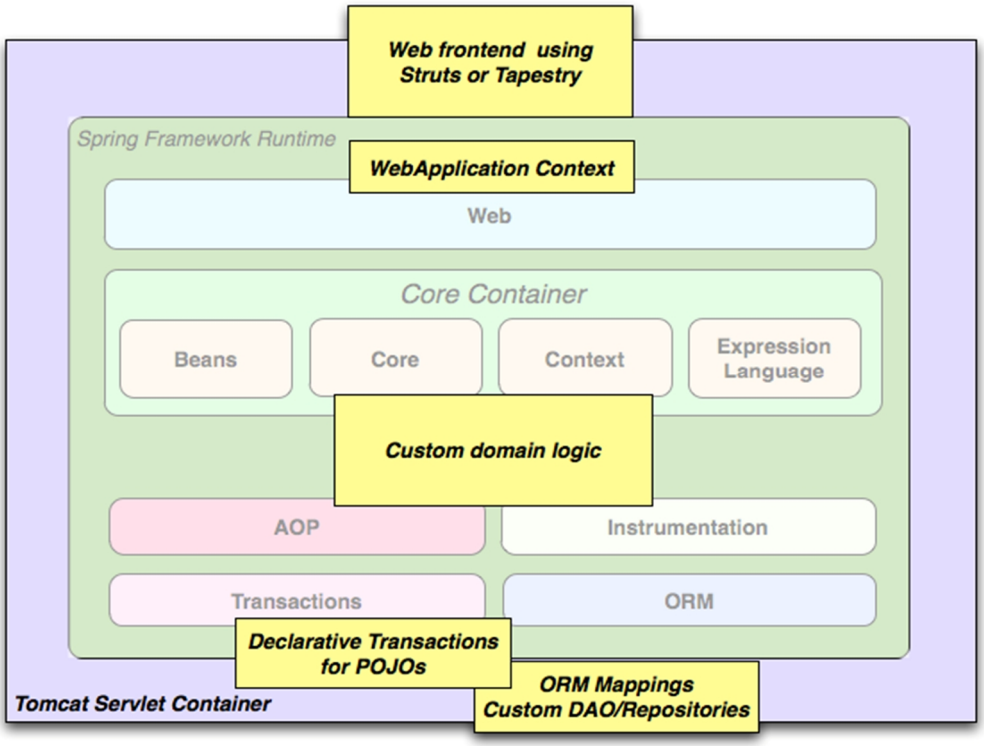


Figura 12. Diagrama de una aplicación Spring con un framework externo

Cuando se necesita obtener acceso al código existente a través de los servicios web, se puede usar Spring de Hesse- , Burlap- , RMI- o las clases JaxRpcProxyFactory. Estos métodos de accesos a datos permiten habilitar fácilmente el acceso remoto a las aplicaciones existentes. A continuación se muestra un diagrama de la organización interna de Spring en un escenario similar al explicado anteriormente.

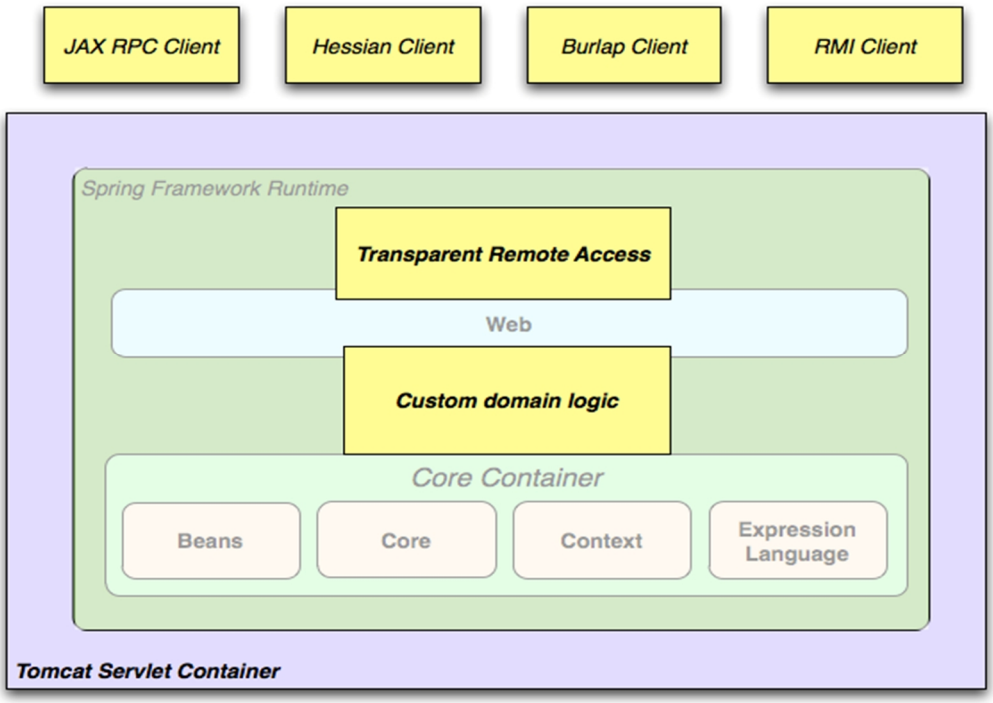


Figura 13. Diagrama de una aplicación Web de Spring

Spring Framework también proporciona una capa de abstracción de acceso para Enterprise JavaBeans (EJB), lo que le permite reutilizar sus POJOs existentes y "envolverlos" en beans de sesión sin estado, para su uso en aplicaciones web escalables que puedan necesitar seguridad declarativa, manteniéndose a prueba de fallos. En el diagrama que se encuentra a continuación se muestra como el Enterprise JavaBeans "envuelve" los POJOs.

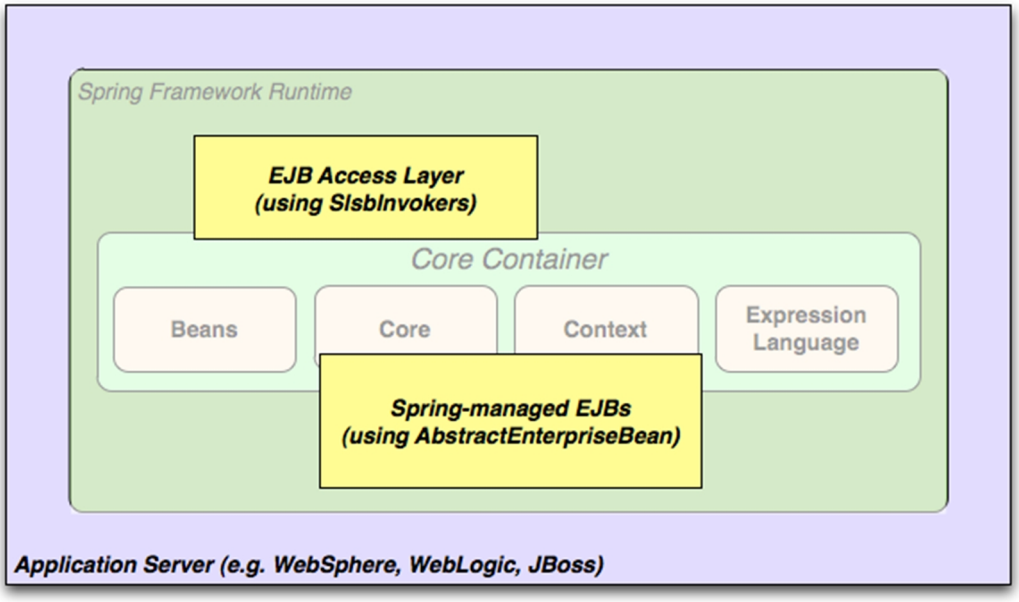


Figura 14. Diagrama de EJB y POJOs

3.1. Gestión de Dependencias y Nomenclatura para las Bibliotecas

La gestión de dependencias y la inyección de dependencias son cosas diferentes. Para conseguir en una aplicación todas las características de Spring se necesitan ensamblar todas las librerías necesarias (archivos .jar) y añadirlas a la ruta de clases (classpath) en tiempo de ejecución y, posiblemente, en tiempo de compilación. Estas dependencias no son componentes virtuales que se inyectan, son por lo general son recursos físicos en un sistema de archivos. El proceso de gestión de la dependencia implica la localización de esos recursos, almacenarlos y agregarlos a la ruta de clases. Las dependencias pueden ser directas (por ejemplo, si la aplicación depende de Spring en tiempo de ejecución), o indirecta (por ejemplo, si la aplicación depende de commons-DBCP que a su vez depende de commons-pool). Las dependencias indirectas también se conocen como "transitivas" y son las más difíciles de identificar y gestionar.

Para desarrollar aplicaciones con Spring se necesita obtener una copia de las bibliotecas .jar que componen las piezas de Spring que se van a necesitar. Para hacer esto más fácil Spring se empaqueta como un conjunto de módulos que separa las dependencias tanto como sea posible, así que por ejemplo, si no se quiere escribir una aplicación web no se necesitan los módulos de Spring Web. Para hacer referencia a los módulos de las bibliotecas de Spring se utiliza una nomenclatura abreviada *spring-** o *spring-*.jar*, donde * representa el nombre corto del módulo (por ejemplo, *Spring-core, Spring-webmvc, Spring-jms*, etc.). El nombre del archivo .jar real que se utiliza es normalmente el nombre del módulo seguido del número de versión (por ejemplo, *spring-core-4.0.2.RELEASE.jar*).

Cada versión de Spring Framework publica artefactos en los siguientes lugares:

- Maven Central, que es el repositorio por defecto de consultas de Maven, y no requiere ninguna configuración especial para su uso. Muchas de las bibliotecas comunes de las que Spring también depende están disponibles en Maven Central, de la misma forma una gran parte de la comunidad de Spring utiliza Maven para la gestión de la dependencia. Los nombres de los .jar de este repositorio tienen la forma de spring-*-<version>.jar, y el groupId de Maven es org.springframework.
- En un repositorio público de Maven organizado específicamente para Spring. Además de las versiones finales GA, este repositorio también alberga hitos e instantáneas del proceso de desarrollo. Los nombres de los archivos .jar tienen la misma forma que Maven Central, por lo que es un lugar muy útil para obtener versiones de desarrollo de Spring para su uso con otras bibliotecas desplegadas en Maven Central. Este depósito también contiene un archivo de distribución de paquetes .zip que contiene todos los .jar de Spring agrupados para facilitar su descarga.

Así que lo primero que hay que decidir es cómo manejar las dependencias. Por lo general, se recomienda el uso de un sistema automatizado como Maven, Gradle o Ivy, pero también se puede hacer de forma manual mediante la descarga de todos los .jar.

Dependencias y funciones de Spring

Aunque Spring ofrece integración y soporte para empresas y otras herramientas externas, mantiene intencionalmente sus dependencias obligatorias a un nivel mínimo para que los usuarios no tengan que localizar y descargar (incluso de forma automática) un gran número de bibliotecas con el fin de utilizar Spring. Para la inyección de dependencia básica sólo hay una dependencia externa obligatoria, y es la de iniciar la sesión.

A continuación se analizarán los pasos básicos necesarios para configurar una aplicación que depende de Spring, en primer lugar con Maven, luego con Gradle y finalmente utilizando Ivy.

Gestión Maven Dependencia

Si se utiliza Maven para la gestión de la dependencia ni siquiera se necesita abastecer la dependencia externa del inicio de sesión, las dependencias de Maven deberían de ser así:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.0.2.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

El ejemplo anterior funciona con el repositorio Maven Central. Para usar el repositorio Maven Spring (por ejemplo, para obtener los hitos y las instantáneas del proceso de desarrollo), se necesita especificar la ubicación del repositorio en la configuración de Maven.

Para las versiones completas:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.release</id>
        <url>http://repo.spring.io/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

Para los hitos:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.milestone</id>
        <url>http://repo.spring.io/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

Y para las instantáneas del proceso de desarrollo:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.snapshot</id>
        <url>http://repo.spring.io/snapshot/</url>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>
</repositories>
```

“Lista de materiales” de dependencias con Maven

También es posible mezclar accidentalmente diferentes versiones de Spring JAR al utilizar Maven. Por ejemplo, es posible que una biblioteca de terceros, o de otro proyecto de Spring, tenga una dependencia transitiva de una versión anterior si se olvida declarar explícitamente una dependencia directa.

Para superar tales problemas Maven apoya el concepto de una "lista de materiales" (BOM) de dependencias. De esta forma se puede importar el spring-framework-bom en la sección DependencyManagement del proyecto para asegurarse de que todas las dependencias de Spring (tanto directas como transitivas) están en la misma versión.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>4.0.2.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
```

```
</dependencyManagement>
```

Un beneficio adicional de la utilización de la "lista de materiales" es que ya no se tendrá que especificar el atributo <version> cuando se dependa de los artefactos de Spring Framework:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>
<dependencies>
```

Gestión Gradle Dependencia

Para usar el repositorio de Spring con el sistema de compilación Gradle, es necesario incluir la URL correspondiente en la sección de repositorios:

```
repositories {
    mavenCentral()
    // and optionally...
    maven { url "http://repo.spring.io/release" }
}
```

Se puede cambiar la URL del repositorio de /release a /milestone o /snapshot, según corresponda. Una vez que el repositorio se ha configurado, se pueden declarar las dependencias siguiendo el proceso habitual con Gradle:

```
dependencies {
    compile("org.springframework:spring-context:4.0.2.RELEASE")
    testCompile("org.springframework:spring-test:4.0.2.RELEASE")
}
```

Gestión Ivy Dependencia

Si se prefiere usar Ivy para gestionar las dependencias existen unas opciones de configuración similares a las vistas anteriormente.

Para configurar Ivy de manera que apunte al repositorio de Spring es necesario agregar el siguiente resolver a ivysettings.xml:

```
<resolvers>
    <ibiblio name="io.spring.repo.maven.release"
            m2compatible="true"
            root="http://repo.spring.io/release/"/>
</resolvers>
```

Se puede modificar la URL de /release a /milestone o /snapshot, según corresponda.

Una vez realizada la configuración, se pueden agregar dependencias siguiendo el proceso habitual. Por ejemplo (en ivy.xml):

```
<dependency org="org.springframework"
name="spring-core" rev="4.0.2.RELEASE" conf="compile->runtime"/>
```

Distribución de los Archivos Zip

Aunque la forma recomendada de utilizar Spring Framework es mediante el uso del sistema de construcción que apoya la gestión de dependencias, también es posible descargar un archivo zip de distribución.

Los zip de distribución se publican en el repositorio Maven Spring (aunque no es necesario Maven o cualquier otro sistema de construcción con el fin de descargarlos).

Para descargar un zip de distribución en necesario entrar en la página http://repo.spring.io/release/org/springframework/spring y seleccionar la subcarpeta correspondiente a la versión que se desee. Las distribuciones también contienen los hitos y las instantáneas del proceso de desarrollo.

3.2. Inicio de sesión

El inicio de sesión o loggin es una dependencia muy importante para Spring porque:

- Es la única dependencia externa obligatoria.
- A todo el mundo le gusta ver las salidas de las herramientas que están utilizando.
- Spring se integra con otras herramientas que también tienen una dependencia de registro.

Uno de los objetivos que un desarrollador de aplicaciones suele tener es el de tener unificado el loggin en un lugar central para toda la aplicación, incluyendo todos los componentes externos.

La dependencia del loggin obligatorio en Spring viene dada por la API Jakarta Commons Logging (JCL). Spring recompila JCL y también hace los objetos JCL Log visibles para todas las clases que extienden de Spring Framework. Uno de los aspectos más importantes para los usuarios es que todas las versiones de Spring utilizan la misma biblioteca de registro, esto facilita la migración debido a que la compatibilidad hacia atrás se conserva incluso con aplicaciones que extienden Spring. La forma de hacer que esto sea posible es permitiendo que uno de los módulos en Spring dependa explícitamente de commons-logging (la aplicación canónica de JCL) y, a continuación, hacer que todos los demás módulos dependan de el en tiempo de compilación.

Lo bueno de commons-logging es que el desarrollador no necesita nada más para desarrollar la aplicación. Cuenta con un algoritmo de descubrimiento de ejecución que busca otros

frameworks de registro situados en lugares bien conocidos de la ruta de clases para utilizar el que considere más apropiado, o el que se le indique.

No utilizar Commons Logging

Desafortunadamente, el algoritmo de descubrimiento de ejecución de commons-logging es problemático. Si se pudiera volver a empezar Spring como un nuevo proyecto usaría una dependencia de registro diferente. Probablemente la primera opción sería la fachada de registro de Java (SLF4J), que también es utilizada por una gran cantidad de herramientas que utilizan las personas con Spring dentro de sus aplicaciones.

Básicamente, hay dos formas para apagar commons-logging:

1. Excluir la dependencia del módulo Spring-core (ya que es el único módulo que depende explícitamente de commons-logging).
2. Crear una dependencia especial de commons-logging que sustituyera a la biblioteca con un .jar vacío.

Para excluir commons-logging, hay que añadir el siguiente fragmento de código a la sección DependencyManagement:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.0.2.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

Ahora bien, esta aplicación probablemente estará "rota" ya que no existe ninguna implementación de la API de JCL en la ruta de clases, por lo que para arreglarlo es necesario proporcionar una nueva.

En las siguientes secciones se mostrará cómo proporcionar una implementación alternativa de JCL usando SLF4J y Log4J.

Usando SLF4J

SLF4J es una dependencia más limpia y eficiente en tiempo de ejecución que commons-logging, ya que utiliza enlaces en tiempo de compilación en lugar de descubrimientos en tiempo de ejecución de los otros frameworks de registro que se integran en la aplicación. Esto también significa que el desarrollador tiene que ser más explícito acerca de lo que quiere que ocurra en tiempo de ejecución, y por lo tanto declararlo o configurarlo en consecuencia. SLF4J proporciona enlaces a muchos frameworks de registro comunes, por lo que normalmente se

puede elegir uno que ya se esté utilizando, con lo que se conseguirá unificar la configuración y la administración.

SLF4J proporciona enlaces a muchos frameworks de registro comunes, incluyendo JCL, y también hace lo contrario, puentes entre otros frameworks de registro y el suyo. Así que para usar SLF4J con Spring es necesario sustituir la dependencia de commons-logging por un puente SLF4J-JCL. Una vez que haya hecho esto entonces el registro de llamadas desde el interior de Spring será traducido al registro de llamadas de la API SLF4J, así que si otras bibliotecas de la aplicación utilizan esa API, entonces se consigue tener solo un lugar para configurar y administrar el registro.

Una opción común es la de hacer un puente de Spring a SLF4J y, a continuación, proporcionar un enlace explícito de SLF4J a Log4J. Para ello se deberán proporcionar 4 dependencias (el puente, la API SLF4J, la unión a Log4J y la implementación Log4J en sí misma) y excluir commons-logging. En Maven habría que hacer algo como esto:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.0.2.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

Aunque pueda parecer que hay que especificar muchas dependencias es opcional, y proporciona un mejor comportamiento que el de commons-logging con respecto a las cuestiones de cargar las clases, sobre todo si se encuentra en un recipiente estricto como la

plataforma OSGi. Además proporciona mejoras en el rendimiento debido a que los enlaces se realizan en tiempo de compilación y no en tiempo de ejecución.

Una opción más común entre los usuarios SLF4J, que utilizan menos pasos y generan un menor número de dependencias, es vincularlo directamente a Logback. Esto elimina un paso de unión extra porque Logback implementa directamente SLF4J, por lo que sólo tendrá que depender de dos bibliotecas y no de cuatro (JCL-over-slf4j y logback). Si se hace esto es necesario también excluir la dependencia slf4j-api de las otras dependencias externas (no de la de Spring), porque sólo se necesita una versión de esa API en la ruta de clases.

Usando Log4J

Mucha gente utiliza Log4j como un framework de registro para fines de configuración y administración. Esto es eficaz y está bien establecido, y de hecho es lo que se suele usar a la hora de desarrollar y probar Spring. Spring también proporciona algunas herramientas para configurar e inicializar Log4j, por lo que tiene una dependencia en tiempo de compilación con Log4j en algunos módulos.

Para hacer el trabajo Log4j con commons-logging todo lo que se necesita hacer es poner Log4j en la ruta de clases y dotarla de un archivo de configuración (log4j.properties o log4j.xml en la raíz de la ruta de clases). Así que para los usuarios de Maven la declaración de dependencia debería ser:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.0.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

A continuación se muestra un ejemplo de log4j.properties para el registro mediante consola:

log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n
log4j.category.org.springframework.beans.factory=DEBUG

Runtime Recipientes con JCL Nativo

Muchas personas ejecutan sus aplicaciones de Spring en un contenedor que a su vez proporciona una implementación de JCL. IBM Websphere Application Server (WAS) es el arquetipo. Esto a menudo causa problemas, y por desgracia no hay una solución mágica, simplemente excluyendo commons-logging de la aplicación no es suficiente en la mayoría de las situaciones.

Para ser claro, los problemas por lo general no están relacionados con JCL, o con commons-logging, sino que tienen que ver con la unión de commons-logging con otro framework (a menudo Log4J). Esto puede ocurrir debido a que commons-logging ha cambiado la forma de hacer el descubrimiento en tiempo de ejecución de cómo lo hacía en las versiones anteriores. Spring no utiliza partes inusuales del API de JCL, cosa que no causa ningún tipo de problema, pero tan pronto como Spring o la aplicación desarrollada traten de hacer cualquier registro se pueden encontrar con que los enlaces a Log4J no están funcionando.

En estos casos la cosa más fácil de hacer es invertir la jerarquía del cargador de clases para que la aplicación sea la que controle la dependencia JCL y no el contenedor. Esa opción no siempre es factible, pero hay muchas otras alternativas que pueden variar dependiendo de la versión exacta y el conjunto de características del contenedor.

PERSISTENCIA DE DATOS

En este capítulo se explicaran las múltiples opciones que Spring proporciona para construir una capa de persistencia. Spring es compatible con todos los frameworks de persistencia que existen para Java como pueden ser Hibernate, Java Persistence API (JPA), iBATIS, o cualquier otro.

1. El acceso a datos

1.1. Patrón DAO

El problema que viene a resolver este patrón es el de contar con varias fuentes de datos, como pueden ser bases de datos, archivos, sistemas externos, etc. Por lo que su principal funcionalidad es la de encapsular la forma en la que se accede a la fuente de datos. Este patrón surgió de la necesidad de gestionar una gran cantidad de fuentes de datos, aunque su uso no solo se extiende a esto, también oculta la forma de acceder a estos datos, de manera que el sistema se centre en los datos que necesita y se olvide de cómo se realiza el acceso a los datos o cual es la fuente de almacenamiento.

Las aplicaciones pueden utilizar el API JDBC para acceder a los datos de una base de datos relacional. Esta API permite una forma estándar de acceder y manipular los datos en una base de datos relacional. También permite a las aplicaciones a las aplicaciones JEE utilizar sentencias SQL, que son el método estándar para acceder a tablas y vistas.

Un DAO define la relación entre la lógica de presentación y empresa por una parte y por otra los datos. El DAO tiene una interfaz común, sea cual sea el modo y fuente de acceso a los datos. Además, proporcionan un medio para leer y escribir los datos en una base de datos, por lo que se debería exponer esta funcionalidad a través de una interfaz para que el resto de la aplicación accediese a ella, de manera que los objetos de servicio no estén acoplados a una única implementación.

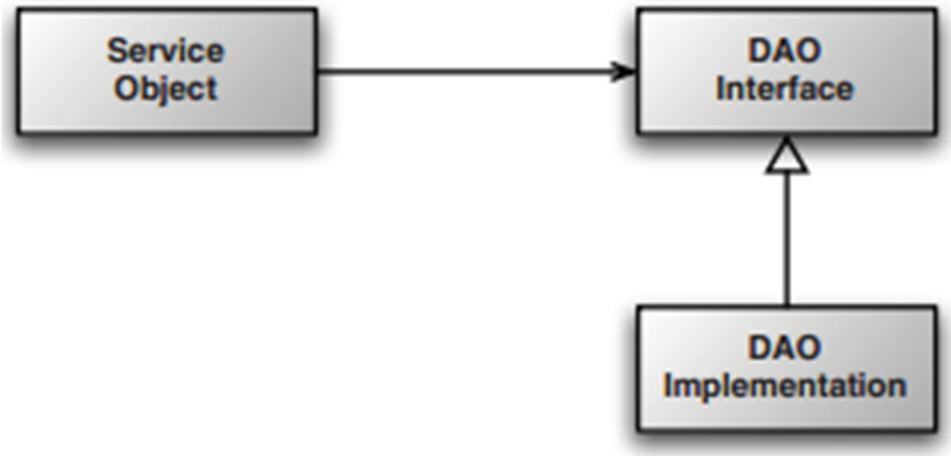


Figura 15. Diagrama de la estructura del patrón DAO

Los objetos de servicio acceden mediante interfaces DAO. Así los objetos de servicio son comprobables y no están acoplados a una única implementación.

1.2. Excepciones en Spring

Al escribir código JDBC estamos obligados a capturar excepciones SQLException, que han podido producirse por diversas razones. Otro problema es que las mayorías de las excepciones provocan una situación fatal que no se puede tratar en el bloque catch. Spring es consciente de todo esto, por lo que el JDBC de Spring proporciona una jerarquía más amplia de excepciones con la que podemos resolver una mayor cantidad de problemas. A continuación se muestra una comparativa de las excepciones que se obtienen mediante el JDBC de Spring y uno normal.

Excepciones JDBC	Excepciones de Spring
BatchUpdateException	CannotAcquierLockException
DataTruncation	CannotSerializaTransactionException
SQLException	CleanupFailureDataAccessException
SQLWarning	ConcurrencyFailureDataAccessException
	DataAccessException
	DataAccessResourceFailureException
	DataIntegrityViolationException
	DataRetrievalFailureException
	DeadlockLoserDataAccessException
	EmptyResultDataAccessException
	IncorrectUpdateSemantisDataException
	InvalidDataAccessResourceUsageException
	OptimisticLockingFailureException
	PermisionDeniedDataAccessException
	PessunustucLockingFailureException
	TypeMismatchDataAccessException
	UncategotizedDataAccessException

Figura 16. Conparativa de excepciones JDBC y Spring.

Todas las excepciones tienen como padre a DataAccessException, la cual es una excepción sin comprobación, con lo que no se tiene que atrapar nada si no se desea. Spring da la libertad al programador para capturarla o no, ya que en otras ocasiones éste se ve obligado a capturar un sinfín de bloques try-catch que a menudo se dejan vacíos.

2. Plantillas

Un método de plantilla define el esqueleto de un proceso que es fijo en una operación. Así las plantillas se responsabilizan de una serie de acciones comunes y devuelven el control a la retrollamada. Spring separa las partes fijas y las variables del proceso de acceso a datos en dos clases distintas: las plantillas y las retrollamadas.

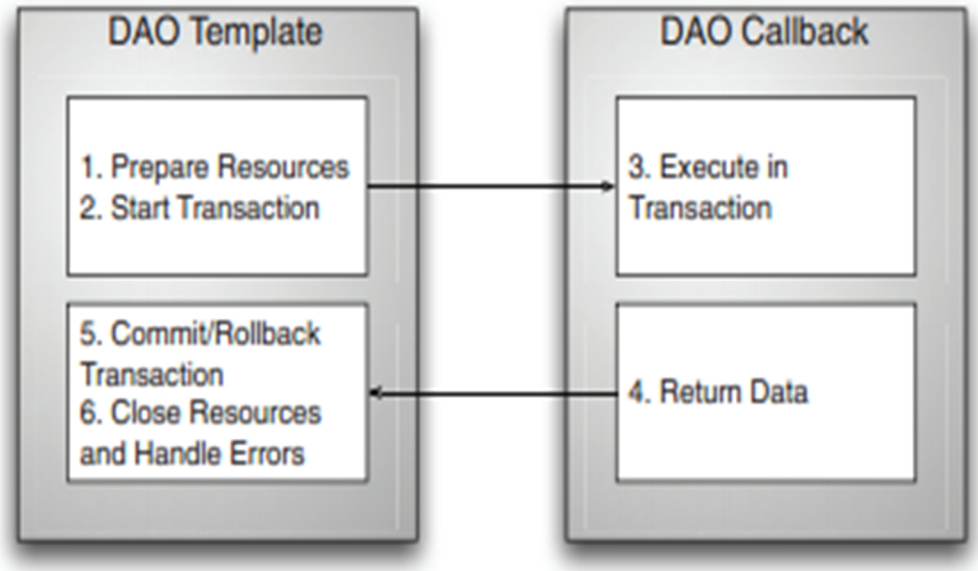


Figura 17. Ciclo de vida de las plantillas DAO

Las plantillas gestionan las partes fijas del acceso a datos, controlan las excepciones, los recursos y manejan las transacciones. Spring tiene varias plantillas dependiendo de la persistencia que vayamos a usar. Usar una plantilla simplifica mucho el código de acceso a datos y se configura como un vean del contexto de Spring.

Plantilla	**Descripción**
CciTemplate	Conexiones JCA CCI
JdbcTemplate	JDBC con soporte para parámetros nombrados
SimpleJdbcTemplate	JDBC simplificadas
HibernateTemplates	Hibernate 2
HibernateTemplate (…orm.hibernate3.*)	Hibernate 3
SqlMapClientTemplate	iBATIS SqlMap
JdoTemplate	JDO
JpaTemplate	JPA Java Persistence Api
TopLinkTemplate	TopLink de Oracle

Figura 18. Tabla de las plantillas existentes

También se pueden usar las clases DAO de Spring para simplificar más la aplicación. Hay clases básicas DAO de Spring que pueden gestionar la plantilla por nosotros.

Uso de clases de soporte DAO

Cada plantilla ofrece métodos prácticos que simplifican el acceso a datos sin necesidad de crear una implementación de retrollamada explicita. Spring proporciona clases de soporte DAO destinadas a ser subclases de las DAO que hagamos en nuestro proyecto.

En la siguiente figura se muestra la relación.

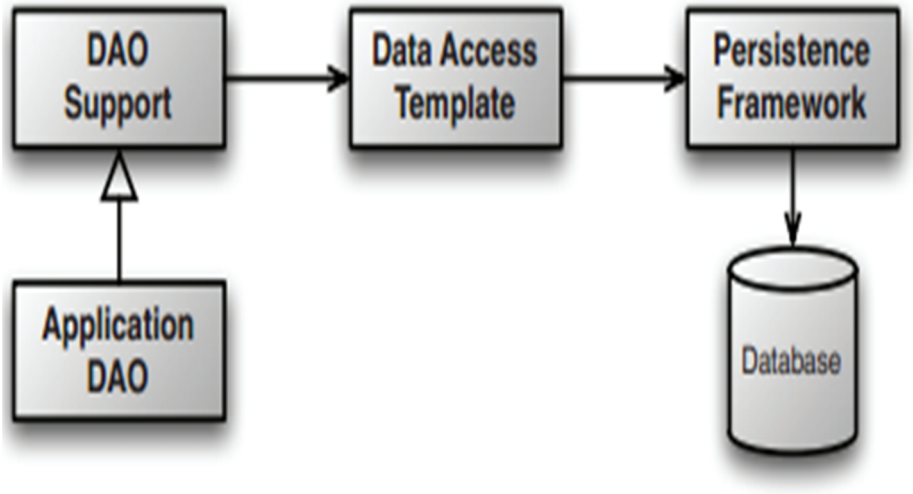


Figura 19. Relación de las plantillas en la aplicación.

Spring ofrece varias clases de soporte DAO. Cada plantilla tendrá su soporte DAO de Spring. En la siguiente tabla se enumeran.

Soporte DAO	Tipo
CciDaoSupport	JCA CCI
JdbcDaoSupport	JDBC
NamedPArameterJdbcDaoSupport	JDBC con soporte para parámetros nombrados
HibernateDaoSupport	Hibernate 2
HibernateDaoSupport	Hibernate 3
SqlMapClientDaoSupport	iBATIS
JdoDaoSupport	JDO
JpaDaoSupport	JPA
TopLinkDaoSupport	TopLink

Figura 20. Soporte DAO en Spring.

3. Configuración de una fuente de datos JDBC

Es la fuente de datos más simple. Spring tiene don clases de fuentes de datos que son:

- **DriverManagerDataSource** que devuelve una nueva conexión cada vez.
- **SingleConnectionDataSource** que devuelve la misma conexión.

El siguiente fragmento de código muestra cómo se configura un **DriverManagerDataSource** en el fichero xml.

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

```
<context:property-placeholder location="jdbc.properties"/>
```

Como se puede ver en el código anterior se tiene que configurar las propiedades url que será la url de acceso a los datos, el usuario con su contraseña y el driver que usara para conectarse a la base de datos.

4. Plantillas JDBC

Todo aquel que haya trabajado con JDBC sabrá que, a pesar del potencial que tiene, para hacer cualquier operación sencilla se tendrá que escribir una innumerable clase con bloques try-catch obligatorios en los que poco se puede hacer. Por ello Spring trata todo este tipo de código estándar repetitivo. Spring limpia el código JDBC asumiendo la gestión de recursos y excepciones. Spring abstrae el código estándar a través de clases plantilla. Hay tres jdbcTemplates, NamedPararmeterJdbcTemplate y SimpleJdbcTemplate.

4.1. JdbcTemplate

Es la implementación más sencilla, permite el acceso a datos mediante JDBC y realizar consultas sencillas de parámetros indexados. Tan solo se necesita un DataSource para funciona, así el código xml será muy sencillo. A partir del DataSource definido en el apartado anterior.

```
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSourde" ref="dataSourceJDBC"/>
</bean>
```

La propiedad DataSource puede ser cualquier implementación de javax.sql.DataSource.

A continuación se muestra un ejemplo con la interfaz persona, en la que se tendrán los métodos get y set para los parámetros id y nombre.

```
Package es.tfg.jdbcspringexample.model
public interface Persona {

        String getNombre();
        void setNombre();
        int getId();
        void setId();
}
```

Una implementación de la clase persona podría ser la siguiente:

```
public class PersonaImp implements Persona {

        private String nombre;
        private int id;
        public int getId(){return id;}
        public int getNombre(){return nombre;}
        public void getNombre(String nombre){ this.nombre=nombre;}
        public void getId(int id){ this.id=id;}
}
```

Esta clase tan sencilla servirá para poder ilustrar la importante labor de los DAO. A continuación se definirá una interfaz DAO:

```
public interface PersonaDao {

        Persona getPersona(int pernosaID);
        List<Persona> getAllPersona();
        void savePersona(Persona persona);
}
```

A continuación se muestra la implementación JDBC para nuestro DAO, el código es el siguiente:

```
public class JdbcPersonaDao implements PersonaDao {

        private JdbcTemplate jdbcTemplate;
        private static final String PERSON_INSERT=
                "insert into persona(id, nombre)"+ "values(?,?)";
        private static final String PERSON_SELECT=
                "select id, nombre from persona";
        private static final String PERSON_SELECT_ID=
                PERSON_SELECT+"where id=?";

        public JdbcPersonDao(JdbcTemplate jdbcTemplate){
                this.jdbcTemplate= jdbcTemplate;}

        public set JdbcPersonDao(JdbcTemplate jdbcTemplate){
                this.jdbcTemplate= jdbcTemplate;}

        public JdbcTemplate getJdbcPersonDao(){
                return jdbcTemplate;}


        public Persona getPersona(int PersonaID){
                List matches = jdbcTemplate.query(PERSON_SELECT_ID,
                        new Object[]{Long.valueOf(personaID)},
                        new RowMapper(){
                        public Object mapRow(ResultSet rs,  int rowNum)
                                throws SQLException,DataAccessException{
                             Persona persona=new PersonaImp();
                             persona.setId(rs.getInt(1));
                            persona.setNombre(rs.getString(2));
                            return persona;
```

```
                        }
                        });
                return matches.size()>0?(Persona)matches.get(0):null;
        }

        public void SavePersona(Persona persona){
                jdbcTemplate.update(PERSON_INSERT, new Object[]{
                    new Integer(persona.getId()),persona.getName()});
        }

        public List<Persona> getAllPersona(){
                List matches = jdbcTemplate.query(PERSON_SELECT,
                        new RowMapper(){
                        public Object mapRow(ResultSet rs,  int rowNum)
                               throws SQLException,DataAccessException{
                            Persona persona=new PersonaImp();
                            persona.setId(rs.getInt(1));
                            persona.setNombre(rs.getString(2));
                            return persona;
                        }
                        });
                return matches;
        }
}
```

Como se puede ver esta clase sirve de plantilla JDBC, la cual inyectaremos desde la definición de Spring con el vean que se ha definido anteriormente. La definición es sencilla:

```
<bean id="jdbcPersonaDao"
class="es.tfg.jdbcspringexample.dao.JdbcPersonaDao">
    <constructor-arg ref="JdbcTemplate"/>
</bean>
```

Analizando el método savePerson vemos que es muy intuitivo y sencillo, gracias a la plantilla que inserta una cadena que contiene una sentencia SQL y un Array de objetos que responden a cada una de las interrogaciones de la sentencia.

La obtención de datos, en los métodos getPersona y getAllPersona son también simples, el uso del objeto RowMapper y el uso de la plantilla del método query. El query toma por parámetro la propia query en cadena, una matriz con los objetos que inserta en cada '?' que se encontrará en la cadena y un objeto RowMapper.

El objeto RowMapper es abstracto y así implementamos el método abstracto rowMap. Este método se ejecutará una vez por fila devuelta en la base de datos.

Ésta implementación está muy bien, pero tiene inconvenientes. Para el método savePersona se utilizan parámetros que han de llevar orden con lo que si algún día hay algún cambio se tendría que tener en cuenta este orden.

4.2. Parámetros nombrados

Es posible usar parámetros nombrados, así se podrá dar a cada parámetro en SQL un nombre explícito. Como por ejemplo:

```
private static final String PERSON_INSERT=
        “insert into persona(id, nombre)”+ “values(:id,:nombre)”;
```

Es decir, se especifica cada nombre correspondiente con el objeto que se vaya a guardar, pero esto no es compatible con JdbcTemplate, así que se creará una nueva clase que en vez de usar un JbdcTemplate use un NamedParameterJdbcTemplate. La clase NamedParameterJdbvPersonDao es un ejemplo y su definición en el fichero xml se hace de la siguiente forma:

```
<bean id="namedJdbcTemplatePersonaDao"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemp
late ">
    <constructor-arg ref="dataSourceJDBC"/>
</bean>
```

```
<bean id="namedJdbcTemplatePersonaDao"
class="es.tfg.jdbcspringexample.dao.NamedParameterJdbcPersonaDao">
    <constructor-arg ref="namedJdbcTemplate"/>
</bean>
```

Para ello se tiene que definir otra plantilla de la clase NamedParameterJdbcTemplate a cuyo constructor tiene hay que pasarle la fuente de datos definida.

4.3. Simplificación del JDBC

A partir de la versión 5 de java es posible pasar listas de parámetros de longitud variable a un método. Así la clase SimpleJdbcPersonDao se codificará de la siguiente manera:

```
public class SimpleJdbcPersonaDao implements PersonaDao {

        private SimpleJdbcTemplate simpleJdbcTemplate;
        private static final String PERSON_INSERT=
                “insert into persona(id, nombre)”+ “values(?,?)”;
        private static final String PERSON_SELECT=
                “select id, nombre from persona”;
        private static final String PERSON_SELECT_ID=
                PERSON_SELECT+“where id=?”;

        public SimpleJdbcPersonDao(SimpleJdbcTemplate
simpleJdbcTemplate){
                this.simpleJdbcTemplate= simpleJdbcTemplate;}
```

```
        public set JdbcPersonDao(SimpleJdbcTemplate
simpleJdbcTemplate){
                this.simpleJdbcTemplate= simpleJdbcTemplate;}

        public SimpleJdbcTemplate getSimpleJdbcPersonDao(){
                return simpleJdbcTemplate;}


        public Persona getPersona(int PersonaID){
              List matches = simpleJdbcTemplate.query(PERSON_SELECT_ID,
                        new ParametrizedRowMapper(){
                        public Object mapRow(ResultSet rs,  int rowNum)
                               throws SQLException {
                             Persona persona=new PersonaImp();
                             persona.setId(rs.getInt(1));
                            person.setNombre(rs.getString(2));
                            return persona;
                        }
                        });
                return matches.size()>0?(Persona)matches.get(0):null;
        }

        public void SavePersona(Persona persona){
               simpleJdbcTemplate.update(PERSON_INSERT, person.getID(),
               persona.getName());
        }

        public List<Persona> getAllPersona(){
                List matches = simpleJdbcTemplate.query(PERSON_SELECT,
                        new ParametrizedRowMapper(){
                        public Object mapRow(ResultSet rs,  int rowNum)
                               throws SQLException,DataAccessException{
                            Persona persona=new PersonaImp();
                            persona.setId(rs.getInt(1));
                            person.setNombre(rs.getString(2));
                            return persona;
                        }
                        });
                return matches;
        }
}
```

Como se puede ver la manera de utilizar esta plantilla es muy similar a la JdbcTemplate, pero ahora se beneficia de las nuevas especificaciones de la versión 5 de Java.

En el método getPersona se puede observar que los parámetros del método query del objeto SimpleJdbcTemplate relativos a la formulación de la query (es decir que parámetros se utilizaran a la hora de ser sustituidos por cada '?') se podrán como los últimos parámetros no teniendo límite y pudiendo ser primitivos (no como en el ejemplo JdbcPersonDao en el cual teníamos que crear objetos Integer), es decir, se podrán hacer autobox. Como s se puede ver en el ejemplo se usa un nuevo objeto abstracto el ParametrizedRowMapper, del cual hay que implementar el método mapRow().

La definición del vean en el fichero xml es la siguiente:

```
<bean id="simpleJdbcTemplate"
class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate ">
    <constructor-arg ref="dataSourceJDBC"/>
</bean>
```

```
<bean id="simpleJdbcPersonaDao"
class="es.tfg.jdbcspringexample.dao.SimpleJdbcPersonaDao">
    <constructor-arg ref="simpleJdbcTemplate"/>
</bean>
```

5. Clases Spring para JDBC

Como se ha podido ver, el código que se ha generado es muy repetitivo. Cada clase DAO tiene que contener una plantilla que a su vez tenga una fuente de datos. Spring proporciona una alternativa para poder disponer de todo esto otorgando clases de las cuales se puede heredar para crear los DAOs.

El siguiente fragmento de código muestra como el Dao extiende de la clase **JdbcDaoSupport**.

```
public class JdbcPersonaDao extends JdbcDaoSupport
                                    implements PersonaDao {

        private static final String PERSON_INSERT=
                "insert into persona(id, nombre)"+ "values(?,?)";
        private static final String PERSON_SELECT=
                "select id, nombre from persona";
        private static final String PERSON_SELECT_ID=
                PERSON_SELECT+"where id=?";


        public Persona getPersona(int PersonaID){
              List matches = getjdbcTemplate().query(PERSON_SELECT_ID,
                        new Object[]{Long.valueOf(personaID)},
                        new RowMapper(){
                        public Object mapRow(ResultSet rs,  int rowNum)
                               throws SQLException,DataAccessException{
                             Persona persona=new PersonaImp();
                             persona.setId(rs.getInt(1));
                            person.setNombre(rs.getString(2));
                            return persona;
                        }
                        });
                return matches.size()>0?(Persona)matches.get(0):null;
        }

        public void SavePersona(Persona persona){
                getjdbcTemplate().update(PERSON_INSERT, new Object[]{
                    new Integer(persona.getId()),persona.getName()});
        }

        public List<Persona> getAllPersona(){
                List matches = getjdbcTemplate().query(PERSON_SELECT,
                        new RowMapper(){
                        public Object mapRow(ResultSet rs,  int rowNum)
```

```
                                throws SQLException,DataAccessException{
                            Persona persona=new PersonaImp();
                            persona.setId(rs.getInt(1));
                            person.setNombre(rs.getString(2));
                            return persona;
                        }
                        });
                return matches;
        }
}
```

Con esto se ha eliminado la tarea de crear métodos get y set para la plantilla. De hecho ni siquiera se tendrá que inyectar una plantilla, sino que al inyectar una fuente de datos el objeto la creará automáticamente.

Al igual que en los anteriores ejemplos también existen clases abstractas de soporte en Spring con estas plantillas. Estas son NamedParameterJdbcDaoSupport y SimpleJdbcDaoSupport.

6. Integración de Hibernate en Spring

En el mercado existen diversas plataformas para dar solución a un sistema de persistencia basado en mapeo de modelos de objeto relacionales (ORM). Spring es compatible con varios frameworks entre ellos se encuentra Hibernate. Spring aporta compatibilidad integrada para transacciones declarativas de Spring, manejo de excepciones, plantillas, soporte DAO y gestión de recursos.

Hibernate es el framework que más acogida ha tenido dentro de la comunidad, siendo este de código abierto y desarrollado en un principio por desarrolladores dispersos alrededor del mundo. No solo otorga ORM, sino que también ofrece soluciones al duplicado de datos, la carga perezosa, eager fetching y duplicado de datos.

Un aspecto importante que hay que tener en cuenta es elegir la versión de Hibernate, al margen de las nuevas funcionalidades, ya que el nombre de los paquetes varía de la versión 2 a la 3 y esto hace que haya que diferenciarlos en Spring. Así para Hibernate 2 se utilizaran las clases contenidas en org.springframework.hibernate, para Hibernate 3 se utilizaran las clases en el paquete org.springframework.hibernate3.

6.1. Instanciar SessionFactory de Hibernate

El objeto LocalSessionFactoryBean de Spring es un bean factory de Spring que produce una instancia local de SessionFactory de Hibernate que toma sus parámetros de mapeo de los archivos XML.

La definición de este bean sería la siguiente:

```
<bean id="localSessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean ">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list>
          <value>es/tfg/hibernateaddendum/model/Exam.hbm.xml</value>
          <value>es/tfg/hibernateaddendum/model/Student.hbm.xml</value>
          <value>es/tfg/hibernateaddendum/model/Subject.hbm.xml</value>
          <value>es/tfg/hibernateaddendum/model/SubjectUnit.hbm.xml
                </value>
        </list>
    </property>
    <property name="hibernateProperties">
        <prop>
          <prop key="hibernate.dialect">${hibernate.dialect}</prop>
        </props>
    </property>
</bean>
```

La propiedad mappingResources define una lista con una entrada por fichero hbm.xml que se definirá para el mapeo del objeto relacional. La propiedad hibernateProperties define propiedades necesarias para la configuración de Hibernate.

6.2. AnnotationSessionFactoryBean

Si lo que se desea es utilizar objetos cuyas clases implementen anotaciones JPA y anotaciones específicas de Hibernate se debe usar la clase AnnotationSessionFactoryBean. Es muy similar a la clase LocalSessionFactoryBean salvo que en vez de hacer una lista con los ficheros hbm.xml la haremos con cada objeto que implemente anotaciones.

```
<bean id="annotationSessionFactoryBean"
class="org.springframework.orm.hibernate3.annotation.AnnotationSession
FactoryBean ">
    <property name="dataSource" ref="dataSource"/>
    <property name="AnnotatedClasses">
        <list>
          <value>es.tfg.hibernateaddendum.model.Exam</value>
          <value>es.tfg.hibernateaddendum.model.Student</value>
          <value>es.tfg.hibernateaddendum.model.Subject</value>
          <value>es.tfg.hibernateaddendum.model.SubjectUnit</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <prop>
          <prop key="hibernate.dialect">${hibernate.dialect}</prop>
        </props>
    </property>
</bean>
```

Como se puede ver en el ejemplo anterior la definición es muy similar a LocalSessionFactoryBean, salvo que hay que tener una propiedad annotatedClasses cuya lista se confecciona con las clases con anotaciones persistentes.

6.3. Hibernate Template

Esta plantilla simplifica el trabajo con el objeto Session de Hibernate, siendo responsable de abrir y cerrar las sesiones y de gestionar las excepciones. Para crear la plantilla hay que pasar una instancia de la FactorySession.

El siguiente fragmento muestra cómo se configura Hibernate Template en Spring:

```
<bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="localSessionFactory"/>
</bean>
```

Y el siguiente fragmento sería para utilizar el annotationSessionFactoryBean definido anteriormente:

```
<bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory"
ref="annotationSessionFactoryBean"/>
</bean>
```

Ambas definiciones son idénticas, cabe señalar que en el atributo sessionFactory se podrá pasarle cualquier clase que implemente FactoryBean.

Una vez hecho todo habrá que crear un DAO que haga uso de la plantilla para que así se pueda persistir y recuperar objetos.

El siguiente código sirve de ejemplo de cómo se hace uso de la plantilla.

```
public class HibernateExamsDaoImp {

        private static final String INSERT="insert";
        private static final String DELETE="delete";
        private static final String GETT_EXAM="getExam";
        private static final String EXAM=EXAM.class.getName();
        private static final String SELECT_ID="from" + EXAM +
                                                "where id = ?";
        private static final String SELECT_ALL="from" + EXAM;
        private HibernateTemplate hibernateRemplate;

        public HibernateTemplate getHibernateTemplate(){
                return hibernateTemplate;
        }

        public void setHibernateTemplate(HibernateTemplate
                                                hibernateTemplate){
                this. hibernateTemplate= hibernateTemplate;
```

```
        }

        public void update(Exam exam){
                getHibernateTemplate().saveorUpdate(exam);
        }

        public void insert(Exam exam){
                getHibernateTemplate().saveorUpdate(exam);
        }

        public void insert(Set<Exam> exams) throws ExamsException{
            for(Iterator<Exam> it = exams.iterator(); it.hasNext();){
                this.insert(it.next());
            }
        }

        public void getExam(int id) throws ExamsException{
            List results = getHibernateTemplate().find(SELECT_ID,id);
            if(results.size()==0){
                return (Exam) results.get(0);
            }
            throws new ExamNotFoundException(id);
        }

        public Collection<Exam> getAllExams() throws ExamsException{
            return getHibernateTemplate().find(SELECT_ALL);
        }

        public void delete(int id) throws ExamsException{
                getHibernateTemplate().delete(this.getExam(id));
        }
}
```

Así las operaciones monótonas necesarias para operar con Hibernate serán controladas por la plantilla.

La definición dentro del contenedor de Spring se hará de la siguiente forma:

```
<bean id="hibernateExamsDaoImp"
class="es.tfg.hibernatespringintegration.dao.HibernateExamsDaoImp">
    <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
```

6.4. HibernateDaoSupport

Al igual que con JdbcDaoSupport, Spring también facilita la construcción de DAO en Hibernate. La clase HibernateDaoSUpport otorga una plantilla HibernateTemplate con lo que sólo tendremos que inyectar un objeto SessionFactory.

El siguiente código pertenece a la clase HibernateStudentsDaoImp que hereda de HibernateDaoSupport:

```
public class HibernateStudentsDaoImp extends HibernateDaoSupport{

        private static final String STUDENT=Student.class.getName();
        private static final String SELECT_ID="from" + STUDENT +
                                                    "where id = ?";
        private static final String SELECT_ALL="from" + STUDENT;

        public void update(Exam student){
                getHibernateTemplate().saveorUpdate(student);
        }

        public void insert(Exam student){
                getHibernateTemplate().saveorUpdate(student);
        }

        public void insert(Set<Student> students)
                                                throws StudentsException{
            for(Iterator<Exam> it = students.iterator();
                                                    it.hasNext();){
                this.insert(it.next());
            }
        }

        public void getStudent(int id) throws ExamsException{
            List results = getHibernateTemplate().find(SELECT_ID,id);
            if(results.size()==0){
                return (Student) results.get(0);
            }
            throws new StudentNotFoundException(id);
        }

        public Collection<Student> getAllStudents()
                                                throws StudentsException{
            return getHibernateTemplate().find(SELECT_ALL);
        }

        public void delete(int id) throws ExamsException{
                getHibernateTemplate().delete(this.getStudent(id));
        }
}
```

Con las clases de soporte Spring se facilita al máximo la codificación de las nuevas clases reduciendo el número de líneas de código de estas y reduciéndolas a la mínima expresión.

7. Java Persistence API (JPA)

Java Persistence API, más conocido como JPA, es la API de persistencia desarrollada para la plataforma Java EE, es un framework de Java que maneja los datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard (Java SE) y Enterprise (EE).

El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto relacional), como pasaba con EJB2, y permitir usar objetos regulares (conocidos como POJOs).

La parte de la especificación de EJB3 que sustituye los beans de entidad se conoce como JPA.

7.1. EntityManagerFactory

Para utilizar JPA hay que usar una implementación de EntityManagerFactory para obtener una instancia de un EntityManager. La especificación JPA define dos tipos de gestores de entidad, gestionados por la aplicación o por el contenedor. Los gestionados por la aplicación se crean directamente cuando la aplicación pide un gestor de entidad a la fábrica de gestores de entidad. Cuando esta se usa la aplicación es la responsable de abrir y cerrar los gestores de entidad y de implicar al gestor de entidad en las transacciones. Sin embargo, si se utiliza un contenedor Java EE la aplicación no interactúa en absoluto con la fábrica de gestor de entidad y es por ello por lo que se debe usar los gestores de entidad gestionados por el contenedor. Así se obtendrán mediante JNDI o mediante inyección. Ambos gestores implementan EntityManager.

A la hora de desarrollo, al utilizar la interfaz, no será necesario saber cual se está utilizando, ya que Spring se encarga de gestionarlos. Para ello Spring tiene dos beans, LocaslEntityManagerFactoryBean y LocalContainerEntityManagerFactoryBean, siendo la primera gestionada por la aplicación y la segunda por el contenedor. Ya que como hemos dicho Spring hace transparente el acceso a ambas, la única diferencia es la forma en la que se definen.

7.1.1. Configurar LocalEntityManagerFactoryBean

En el archivo persistence.xml se hallan la mayor parte de la configuración. En este archivo se definen tantas unidades de persistencia como se deseen, enumerando las persistentes.

```
<?xml version="1.0" encoding=”UTF-8”?>
<persistence xmlns:xsi="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instancee"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
        version="1.0">
<persistence-unit name="CbosCommonsPU"
                                transaction-type="RESOURCE_LOCAL"/>
<provider> org.apache.openjpa.persistence.PersistenceProviderImp
</provider>
        <properties>
                <property
                        name="openjpa.ConnectionDriverName"
                        value="org.postgresql.Driver"/>
                <property
                        name="openjpa.ConnectionURL"
        value="jdbc:posrgresql://192.168.1.214.5432/arcadia_cbos"/>
                <property
                        name="openjpa.ConnectionUserName"
                        value="postgres"/>
                <property
                        name="openjpa.ConnectionPassword"
                        value="postavalon"/>

        <properties>
```

```
<persistence-unit> <persistence>
```

Después se define el bean en el fichero de Spring y se le pasa la unidad de persistencia como parámetro.

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="youdiversity"/>
    <property name="jpaVendorAdapter">
        <bean

class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter">
                <property name="showSql" value="true"/>
                <property name="generateDdl" value="true"/>
                <property name="database" value="HSQL"/>
        </bean>
    </propierty>
</bean>
```

7.1.2. Configurar LocalContainerEntityManagerFactoryBean

Lo primero que hay que hacer para configurar un LocalContainerEntityManagerFactoryBean es obtener una EntityManagerFactory utilizando la información proporcionada por el contenedor. Esta es la configuración típica cuando se usa JBoss o WebLogic, es decir, para servidores de aplicación JEE.

En el siguiente fragmento de código se muestra como configurar un LocalContainerEntityManagerFactoryBean.

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryB
ean">
    <property name="dataSource" ref="youdiversityDataSource"/>
    <property name="jpaVendorAdapter">
        <bean

class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter">
                <property name="showSql" value="true"/>
                <property name="generateDdl" value="true"/>
                <property name="database" value="HSQL"/>
        </bean>
    </propierty>
</bean>
```

Como se puede ver se ha tenido que configurar un bean interno de tipo TopLinkJpaVendorAdapter, ya que la propiedad jpaVendorAdapter puede tener varios valores y hace referencia a la implementación de JPA a utilizar.

7.2. Plantillas JPA

Al igual que con otras soluciones de persistencia, Spring otorga una plantilla llamada JpaTemplate. Esta plantilla contendrá un EntityManager de JPA. En el siguiente xml se puede ver como se configura la plantilla.

```
<bean id="jpaTemplate"
class="org.springframework.orm.jpa.JpaTemplate">
  <property name="entityManagerFactory" ref=" entityManagerFactory "/>
</bean>
```

La propiedad entityManagerFactory de JpaTemplate debe conectarse con una implementación de la interfaz javax.persistence.EntityManagerFactory de JPA. JpaTemplate a, igual que otras plantillas tiene muchos métodos de acceso a datos ofrecidos por un EntityManager nativo. Pero implica a los EntityManager en transacciones y maneja excepciones.

El siguiente DAO es un ejemplo de cómo se utiliza JpaTemplate:

```
import org.springframework.orm.jpa.JpaTemplate;
…
public class UserJpaDao{

        private JpaTemplate jpaTemplate;

        public JpaTemplate getJpaTemplate(){
                return jpaTemplate;
        }

        public void setJpaTemplate(JpaTemplate jpaTemplate){ {
                this.jpaTemplate= jpaTemplate;
        }
…
}
```

Para configurar este DAO simplemente conectamos la JpaTemplate a la propiedad jpaTemplate:

```
<bean id="userDao" class="yourdiversity.model.orm.dao.jpa.UserJpaDao">
  <property name="jpaTemplate" ref="jpaTemplate"/>
</bean>
```

Una manera de usar la plantilla sería:

```
public void saveUser(User user){
        getJpaTemplate().persist(user);
}

public List<User> getAllUsers(){
```

```
        return getJpaTemplate().find("select u from User u;“);}
```

7.3. Extensión de JpaSupport

La clase JpaDaoSupport es una clase abstracta que tiene una plantilla JpaTemplate, así que en vez de conectar a cada DAO su plantilla, lo que se puede hacer es extender de JpaDaoSupport y conectar el vean entityManagerFactory.

Así el Dao extenderá del JpaDaoSupport así:

```
public class BaseDaoJpa<T extends BaseEntity> extends JpaDaoSupport {
…
}
```

La definición sería tan sencilla como:

```
<bean id="baseDao" class="yourdiversity.model.orm.dao.jpa.BaseDaoJpa">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

SPRING MVC

1. Introducción a Spring Framework Web MVC

Spring Framework Modelo-Vista-Controlador (MVC) está diseñado en torno a un DispatcherServlet que envía solicitudes a los controladores, con mapas de control configurables, resolución de vista, resolución de localización, resolución de temas y soporte para la carga de archivos. El controlador predeterminado se basa en @Controller y @RequestMapping, que ofrecen una amplia gama de métodos de control flexibles. El mecanismo @Controller también permite crear aplicaciones y páginas Web a través de anotaciones en la @PathVariable.

En Spring Web MVC se puede utilizar cualquier objeto como un comando o como un formulario, de manera que no se necesita implementar una interfaz específica o una clase. El enlace de datos es muy flexible, por ejemplo, trata a los errores de tipo como errores de validación y no como errores del sistema, de manera que puedan ser evaluados por la aplicación.

La resolución de la vista es extremadamente flexible. El controlador es normalmente el responsable de preparar un modelo Map con los datos y seleccionar un nombre de vista, pero también puede escribir directamente en la secuencia de respuesta y completar la solicitud. La resolución del nombre de la vista es configurable a través de la extensión del archivo mediante los nombres de los beans o incluso mediante la implementación personalizada del ViewResolver. El modelo es un mapa de la interfaz, lo que permite obtener una abstracción completa de la de vista. Se puede integrar la vista directamente con plantillas basadas en las tecnologías de representación tales como JSP, Velocity y Freemarker o directamente generar XML, JSON, Atom, y muchos otros tipos de contenidos. El modelo Map es simplemente transformado en el formato adecuado, como los atributos de la petición JSP o un modelo de plantilla Velocity.

1.1. Características de Spring Web MVC

El módulo de Spring Web incluye muchas características únicas de soporte Web:

- Clara separación de los roles. Cada rol pueden ser cumplido por un objeto especializado.
- Configuración potente y directa tanto del framework y las clases de la aplicación como del JavaBeans. Esto facilita la consulta a través de contextos, como el de los controladores Web para los objetos y los validadores.
- Adaptabilidad, no injerencia y la flexibilidad. Definir cualquier método controlador que se necesite, usando una de las anotaciones de parámetros (@ RequestParam, @ RequestHeader, @ PathVariable, etc.) para un escenario dado.
- Reutilización de código. Utilización de los objetos existentes como comandos o formularios, en lugar de la duplicarlos para extender una clase particular del framework.

- Enlaces y validación personalizados. Trata los errores de tipo como errores de validación a nivel de aplicación que mantienen el valor que lo produjo.
- Asignación de control y resolución de la vista personalizable. Estrategias para la asignación de control y la resolución de las vistas que van desde la configuración en base a URL sencillas, hasta sofisticadas estrategias de resolución de propósitos.
- Transferencia de modelo flexible. Transferencia de modelo con un nombre/valor Map que es integrado fácilmente con cualquier tecnología de vistas.
- Obtención de la configuración regional, de la zona horaria y del tema, soporte para JSP con o sin etiquetas de bibliotecas, soporte para JSTL, soporte para Velocity sin necesidad de puentes.
- Una biblioteca de etiquetas JSP que proporciona soporte para funciones como el enlace de datos y temas.
- Una biblioteca de etiquetas JSP que hace que la escritura en las páginas JSP sea mucho más fácil.
- Beans cuyo ciclo de vida está en el ámbito de la actual HTTPRequest o HTTPSesion. Esto no es una característica específica de Spring MVC, sino más bien del recipiente WebApplicationContext que utiliza Spring MVC.

2. El DispatcherServlet

Spring Web framework MVC es impulsado por una solicitud, esta diseñado en torno a un Servlet central que envía solicitudes a los controladores y ofrece funcionalidades que facilitan el desarrollo de aplicaciones web. Sin embargo el DispatcherServlet de Spring no se limita a eso, está completamente integrado con el contenedor IoC y como tal le permite usar todas las otras características proporcionadas por Spring.

El flujo de procesamiento de solicitudes de Spring Web MVC DispatcherServlet se ilustra en el siguiente diagrama. El DispatcherServlet es una expresión del patrón de diseño Front Controller.

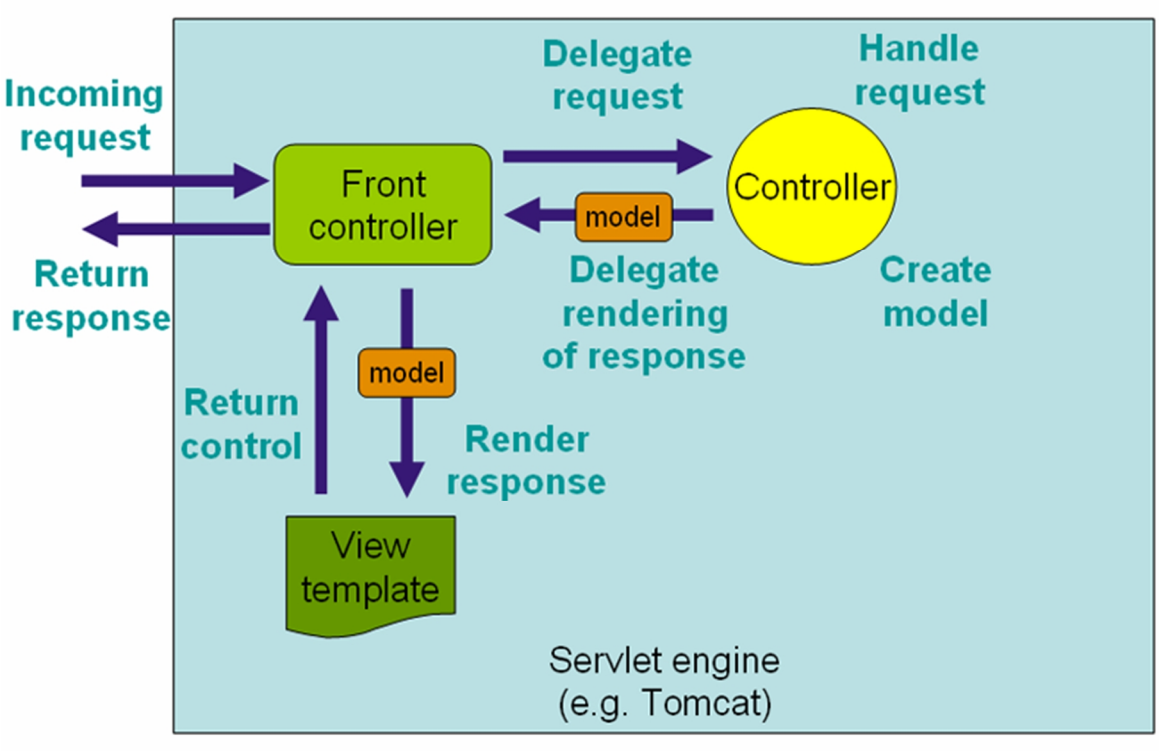


Figura 21. Flujo de procesamiento de solicitudes.

El DispatcherServlet es un Servlet real (hereda de la clase HttpServlet), y como tal se declara en el archivo web.xml de la aplicación web. Es necesario seleccionar las peticiones que el DispatcherServlet vaya a manejar mediante la asignación de una dirección URL en el archivo web.xml. A continuación se muestra un ejemplo de la declaración del DispatcherServlet:

```
<web-app>
    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>/example/*</url-pattern>
    </servlet-mapping>

</web-app>
```

En el ejemplo anterior, todas las peticiones que empiezan por /example son manejadas por la instancia del DispatcherServlet llamada example. A partir de la versión Servlet 3.0 también se puede configurar el contenedor de programación de Servlets. A continuación se muestra el código equivalente al ejemplo anterior:

```
public class MyWebApplicationInitializer implements
WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration =
container.addServlet("dispatcher", new DispatcherServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/example/*");
    }

}
```

WebApplicationInitializer es una interfaz proporcionada por Spring MVC que detecta la configuración basada en código y automáticamente la utiliza para inicializar cualquier contenedor del Servlet. Una implementación de la clase abstracta llamada AbstractDispatcherServletInitializer hace que sea aún más fácil de registrar el DispatcherServlet.

Cada DispatcherServlet tiene su propio WebApplicationContext, este hereda todos los beans que ya están definidos en la raíz del WebApplicationContext. Estos beans se pueden sobrescribir, además de crear nuevos beans en una instancia de un servlet determinado.

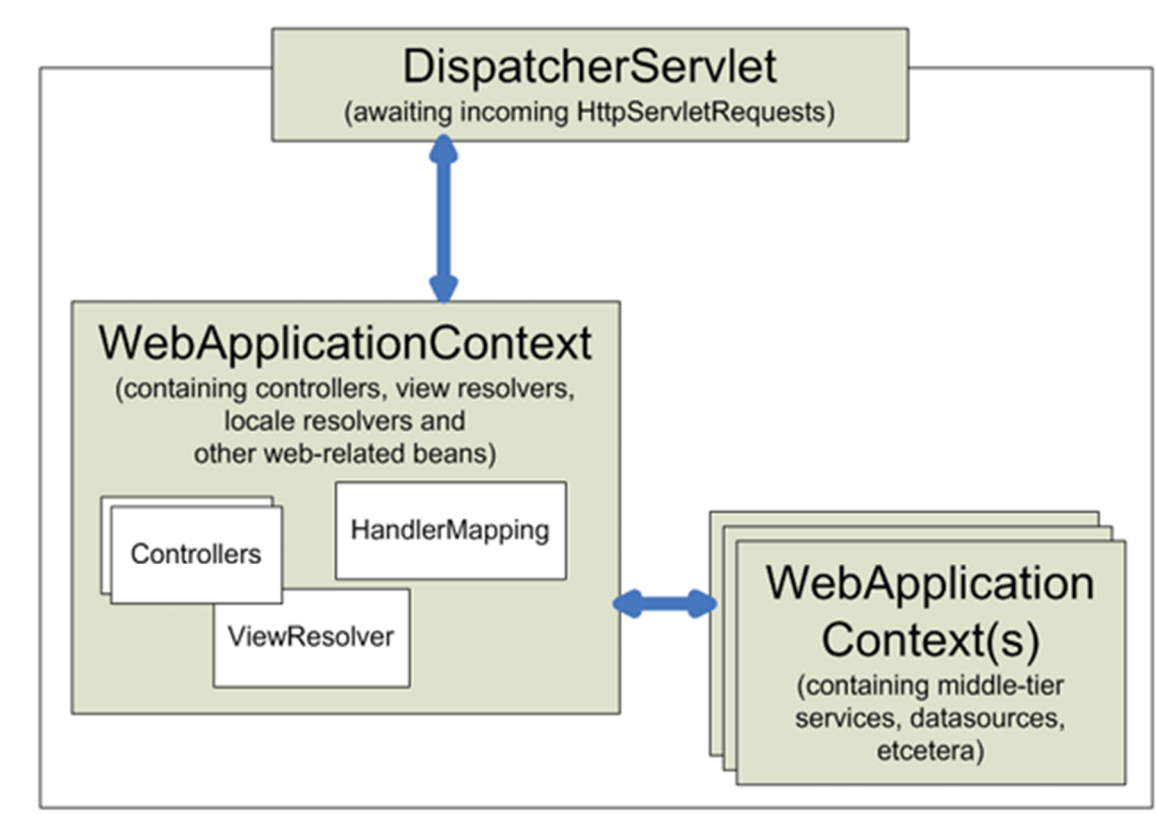


Figura 22. Relaciones del DispatcherServlet.

En la inicialización de un DispatcherServlet, Spring MVC busca un archivo llamado [servlet-name]-servlet.xml en el directorio WEB-INF de la aplicación web y crea los beans definidos

allí, ignorando las definiciones de cualquier bean definido con el mismo nombre en ámbito global.

A continuación se muestra la configuración de un DispatcherServlet en el archivo web.xml:

```
<web-app>
    <servlet>
        <servlet-name>golfing</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>golfing</servlet-name>
        <url-pattern>/golfing/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Con la configuración del servlet anterior, la aplicación tendrá que tener un archivo llamado /WEB-INF/golfing-servlet.xml; este archivo contendrá todos los componentes específicos (beans) de Spring Web MVC.

WebApplicationContext es una extensión de ApplicationContext que tiene algunas características adicionales necesarias para las aplicaciones web. Se diferencia de ApplicationContext en que es capaz de resolver temas y que conoce con que Servlet está asociado, ya que tiene un enlace al ServletContext. WebApplicationContext está unido a un ServletContext, y mediante el uso de métodos estáticos en la clase RequestContextUtils es posible acceder si es necesario al WebApplicationContext.

2.1 Tipos especiales de beans en WebApplicationContext

Spring DispatcherServlet utiliza beans especiales para procesar las solicitudes y representar las vistas adecuadas. Estos beans son parte de Spring MVC. Es posible elegir qué beans utilizar simplemente configurando uno o más de ellos en el WebApplicationContext, sin embargo, no es necesario hacer que Spring MVC mantenga una lista de los beans a utilizar si no se configura ninguna. Antes de profundizar en este tema se mostrara una lista de los tipos de beans especiales en los que se basa el DispatcherServlet.

Tipo de Beans	Explicación
HandlerMapping	Mapa de peticiones entrantes a los controladores y una lista de pre-y post-procesadores en base a unos criterios cuyos detalles varían según la implementación del HandlerMapping.
HandlerAdapter	Ayuda al DispatcherServlet a invocar un controlador asignado a la petición.
HandlerExceptionResolver	Mapas de excepciones para vistas, lo que permite un fácil manejo de excepciones más complejas.
ViewResolver	Resuelve los nombres de vistas basándose en cadenas lógicas dependiendo del tipo de las vistas.
LocaleResolver &LocaleContextResolver	Obtiene la configuración regional que un cliente está utilizando y su zona horaria, con el fin de ser capaz de ofrecer vistas internacionalizadas
ThemeResolver	Resuelve temas que la aplicación web puede utilizar, por ejemplo, para ofrecer diseños personalizados
MultipartResolver	Analiza las solicitudes que se dividen en varias partes, por ejemplo, para apoyar la carga de archivos de procesamiento de formularios HTML.
FlashMapManager	Almacena y recupera la "entrada" y "salida" FlashMap que se utilizan para transferir los atributos de una solicitud a otra, por lo general, a través de una redirección.

Figura 23. Beans del DispatcherServlet.

2.2. Configuración por defecto del DispatcherServlet

Como se mencionó en la sección anterior para cada bean especial del DispatcherServlet se mantiene una lista de las implementaciones a usar por defecto. Esta información se guarda en el archivo DispatcherServlet.properties dentro del paquete org.springframework.web.servlet.

Todos los beans especiales tienen valores por defecto, aunque, tarde o temprano, habrá que personalizar alguna de las propiedades que estos beans proporcionan. Por ejemplo es muy común para configurar un InternalResourceViewResolver y cambiar su propiedad prefix por el directorio de los archivos de la vista.

Una vez que se configura un bean especial, como un InternalResourceViewResolver en el WebApplicationContext, se reemplaza la lista de implementaciones predeterminadas que se han utilizado para ese tipo de bean especial. Por ejemplo, si se configura un InternalResourceViewResolver, se ignorará la lista de implementaciones predeterminadas de ViewResolver.

2.3. Secuencia de procesamiento del DispatcherServlet

Después de configurar un DispatcherServlet, cuando llega una petición para el DispatcherServlet, este empieza a procesar la solicitud de la siguiente manera:

- Se busca el WebApplicationContext y se consolidará la solicitud como un atributo que el controlador y otros elementos en el proceso pueden usar. Está se asocia por defecto en la clave DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE.

- La resolución de la configuración regional se une a la petición para permitir que los elementos obtengan la localización del proceso que se utilizará al procesar la solicitud (la representación de la vista, la preparación de los datos, etc.). Si no se utiliza resolución de la configuración regional este paso se omite.
- La resolución del temas se une con la petición para permitir que los elementos, como las vistas, determinen qué tema usar. Si no se utiliza resolución de temas este paso se omite.
- Si especifica una resolución de archivo de varias partes, la solicitud lo inspecciona; si se encuentran varias partes, la solicitud se envuelve en un MultipartHttpServletRequest para su posterior procesamiento por otros elementos del proceso.
- Se busca un controlador adecuado. Si se encuentra un controlador, la cadena de ejecución asociada con el controlador (pre-procesadores, post-procesadores y controladores) se ejecuta con el fin de preparar un modelo o vista.
- Si se devuelve un modelo, la vista se representa. Si no se devuelve ningún modelo, (puede ser debido a que un pre-procesador o post-procesador intercepte la petición, tal vez por razones de seguridad), la vista no se representa, porque la petición ya se ha cumplido.

La resolución de excepciones de los controladores, es declarada en el WebApplicationContext, recoge excepciones que se producen durante el procesamiento de la solicitud. El uso de estos solucionadores de excepciones permite definir comportamientos personalizados para abordar las excepciones.

Spring DispatcherServlet también permite la devolución de la última fecha de modificación, como se especifica en el API Servlet. El proceso para determinar la última fecha de modificación de una solicitud específica es sencillo: el DispatcherServlet busca y comprueba si el controlador implementa la interfaz LastModified. Si es así, el valor del método getlastmodified (request) de la interfaz LastModified se devuelve al cliente.

También se puede personalizar la instancia del DispatcherServlet añadiendo los parámetros (init-param) de inicialización a la declaración de servlets en el archivo web.xml. La tabla siguiente muestra la lista de parámetros admitidos.

Parámetro	Explicación
contextClass	Clase que implementa WebApplicationContext, lo que crea una instancia del contexto utilizado por este servlet. Por defecto, se utiliza el XmlWebApplicationContext.
contextConfigLocation	Cadena que se pasa a la instancia del contexto (especificado por contextClass) para indicar donde se pueden encontrar los contextos. La cadena consiste potencialmente de varias cadenas (utilizando una coma como delimitador) para soportar múltiples contextos. En caso de utilizar distintos contextos con los beans que se definen en dos ocasiones, la última ubicación tiene prioridad.
espacio de nombres	Espacio de nombres de la WebApplicationContext. El valor predeterminado es [servlet-name]-servlet.

Figura 24. Parámetros admitidos por el DispatcherServlet.

3. Los controladores

Los controladores proporcionan acceso al comportamiento de las aplicaciones que normalmente se definen a través de una interfaz. Los controladores interpretan la entrada del usuario y la transforman en un modelo que se presenta en forma de vista. Spring implementa un controlador de una manera abstracta, lo que le permite crear una amplia variedad de controladores.

En Spring 2.5 se introdujo un modelo de programación basado en anotaciones para los controladores del MVC, esto permite utilizar anotaciones como @RequestMapping, @RequestParam, @ModelAttribute, etc. Estas anotaciones están disponibles tanto para Servlets MVC como para portlets MVC. Los controladores implementados de esta forma no tienen que extender clases o implementar interfaces específicas. Además, por lo general, no tienen dependencias directas con los Servlets o la API de portlets, aunque el acceso a los servlets o portlets se puede configurar fácilmente.

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "helloWorld";
    }
}
```

Como se puede ver, las anotaciones @Controller y @RequestMapping permiten establecer métodos más flexibles. En este ejemplo el método acepta un modelo y devuelve el nombre de la vista a mostrar como un String. @Controller, @RequestMapping y otras anotaciones constituyen la base para la ejecución Spring MVC. En esta sección se documentaran estas anotaciones y se explicara su uso más frecuente en un entorno Servlet.

3.1. Definición de un controlador con @ Controller

La anotación @Controller indica que una clase particular toma el papel de controlador. En Spring, no es necesario extender de ninguna clase el controlador o hacer referencia a la API Servlet. Sin embargo, se puede hacer referencia a características de un Servlet específico.

La anotación @Controller actúa como un marcador que indica las clases en las que se producen anotaciones. El dispatcher busca cada clase con anotaciones y detecta las anotaciones @RequestMapping.

Se pueden definir explícitamente anotaciones para controlar los beans, utilizando una definición de bean estándar en el contexto del dispatcher. Sin embargo, el @Controller también permite la detección automática, detectando las clases de componentes existentes en la ruta de clases y registrando automáticamente las definiciones de los beans.

Para habilitar la detección automática de cada controlador de anotaciones, es necesario añadir la exploración de componentes a la configuración de la aplicación. Para ello hay que usar el spring-context schema, como se muestra en el siguiente fragmento de código XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-
context.xsd">

    <context:component-scan base-
package="org.springframework.samples.petclinic.web"/>

    <!-- ... -->

</beans>
```

3.2. Mapping Request con @RequestMapping

La anotación @RequestMapping se utiliza para asignar direcciones URL como /appointments a una clase completa o a un método controlador en particular. Normalmente las anotaciones a nivel de clase se asignan una ruta de solicitud específica en un controlador, con anotaciones adicionales a nivel de método se reduce de la asignación para métodos específicos de petición HTTP ("GET", "POST", etc.) o para parámetros específicos de condición de la petición HTTP.

El siguiente ejemplo muestra un controlador en una aplicación Spring MVC que utiliza esta anotación:

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

    @Autowired
    public AppointmentsController(AppointmentBook appointmentBook) {
        this.appointmentBook = appointmentBook;
    }

    @RequestMapping(method = RequestMethod.GET)
    public Map<String, Appointment> get() {
        return appointmentBook.getAppointmentsForToday();
    }

    @RequestMapping(value="/{day}", method = RequestMethod.GET)
    public Map<String, Appointment> getForDay(@PathVariable
@DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    @RequestMapping(value="/new", method = RequestMethod.GET)
    public AppointmentForm getNewForm() {
        return new AppointmentForm();
    }

    @RequestMapping(method = RequestMethod.POST)
    public String add(@Valid AppointmentForm appointment,
BindingResult result) {
        if (result.hasErrors()) {
            return "appointments/new";
        }
        appointmentBook.addAppointment(appointment);
        return "redirect:/appointments";
    }
}
```

En el ejemplo, @RequestMapping se utiliza en muchos sitios. La primera aparición es a nivel de clase, lo que indica que todos los métodos de este controlador son en relación a la ruta /appointments. En el método get () la notación @RequestMapping proporciona una mejora importante, de manera que sólo acepta peticiones GET, lo que significa que un HTTP GET para /appointments invoca este método. El método post () tiene un efecto similar y en la getNewForm () combina la definición del método HTTP y de la ruta de acceso en una sola, por lo que las solicitudes GET de appointments/new son manejados por ese método.

El método getForDay () muestra otro uso de @RequestMapping: plantillas URI, que se verá más adelante.

No es necesario @RequestMapping a nivel de clase. Sin esto, todas las rutas son simplemente absolutas y no relativas. El siguiente ejemplo muestra un controlador multi-acción utilizando @RequestMapping:

```java
@Controller
public class ClinicController {

    private final Clinic clinic;

    @Autowired
    public ClinicController(Clinic clinic) {
        this.clinic = clinic;
    }

    @RequestMapping("/")
    public void welcomeHandler() {
    }

    @RequestMapping("/vets")
    public ModelMap vetsHandler() {
        return new ModelMap(this.clinic.getVets());
    }

}
```

Clases de apoyo para los métodos @RequestMapping

Spring 3.1 introdujo un nuevo conjunto de clases de apoyo para los métodos @RequestMapping, llamados RequestMappingHandlerMapping y RequestMappingHandlerAdapter. Las nuevas clases de apoyo están habilitadas por defecto por el espacio de nombres MVC y por la configuración Java MVC, pero se deben configurar de forma explícita si no se va a utiliza ninguno.

Con estas clases de apoyo, el RequestMappingHandlerMapping es el único lugar en el que se elige que método se encargará de procesar la solicitud. Esto permite nuevas posibilidades, por ejemplo ahora un HandlerInterceptor o HandlerExceptionResolver pueden esperar que el controlador basado en objetos se comporte como un HandlerMethod, lo que permite examinar el método exacto, sus parámetros y sus anotaciones. Ya no es necesario dividir el procesamiento de una solicitud en diferentes controladores.

También hay varias cosas que no es posible hacer:

- Seleccionar primero un controlador con SimpleUrlHandlerMapping o BeanNameUrlHandlerMapping y luego reducir el método basado en anotaciones @RequestMapping.
- Asignar los nombres de los métodos como un mecanismo de retroceso para eliminar la ambigüedad entre dos métodos @RequestMapping que no tienen asignada una ruta URL explícita pero que por lo demás coinciden exactamente. En las nuevas clases de apoyo los métodos @RequestMapping tienen que ser asignados de forma exclusiva.
- Tener un único método por defecto (sin una asignación de ruta explícita) con el que las solicitudes se procesen si no hay otro método de control que coincida concretamente. En las nuevas clases de apoyo si no se encuentra un método de emparejamiento se produce un error 404.

Patrones de plantilla URI

Las plantillas URI se pueden utilizar para facilitar el acceso a determinadas partes de una URL en un método @RequestMapping.

Una plantilla URI es una cadena que contiene uno o más nombres de variables. Al sustituir los valores de estas variables, la plantilla se convierte en un URI. Las plantillas URI están definidas cómo un URI parametrizado. Por ejemplo, la plantilla URI http://www.example.com/users/{userId} contiene la variable userId.

En Spring MVC puede utilizar la anotación @PathVariable en un argumento de un método para hacer que tome el valor de una variable de la plantilla URI:

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

La plantilla URI "/owners/{ownerId}" especifica el nombre de la variable ownerId. Cuando el controlador atiende esta solicitud, el valor de ownerId se convierte en el valor del URI. Por ejemplo, cuando llega una petición para /owners/fred, el valor de ownerId es fred.

Un método puede tener cualquier número de anotaciones @PathVariable:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}",
method=RequestMethod.GET)
public String findPet(@PathVariable String ownerId, @PathVariable
String petId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    Pet pet = owner.getPet(petId);
    model.addAttribute("pet", pet);
    return "displayPet";
}
```

Cuando se usa una anotación @PathVariable en un argumento Map<String, String>, el mapa se completa con todas las variables de la plantilla URI.

Una plantilla URI puede ser formada a partir del tipo y de las anotaciones del @RequestMapping. Como resultado, el método findPet () puede ser invocado mediante una URL como /owners/42/pets/21.

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping("/pets/{petId}")
    public void findPet(@PathVariable String ownerId, @PathVariable
String petId, Model model) {
        // implementation omitted
```

```
    }
}
}
```

Un argumento @PathVariable puede ser de cualquier tipo como int, long, etc., ya que Spring lo convierte automáticamente al tipo adecuado o lanza una TypeMismatchException si no lo hace.

Patrones para plantillas URI con expresiones regulares

A veces es necesario tener mayor precisión en la definición de variables de la plantilla URI, para ello la anotación @RequestMapping apoya el uso de expresiones regulares con variables de la plantilla URI. La sintaxis es {nomVar: regex} donde la primera parte se define el nombre de la variable y la segunda la expresión regular. Por ejemplo:

```
@RequestMapping("/spring-web/{symbolicName:[a-z-]}-
{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]}")
    public void handle(@PathVariable String version, @PathVariable
String extension) {
        // ...
    }
}
```

Patrones Path

Además de las plantillas URI, la anotación @RequestMapping también es compatible con los patrones Path Ant-Style (por ejemplo, /myPath/*.do).

Variables de matriz

La especificación URI RFC 3986 define la posibilidad de incluir un par nombre-valor dentro de los segmentos del path. Dentro Spring MVC estos se conocen como variables de matriz.

Las variables de matriz pueden aparecer en cualquier segmento del path, cada variable de matriz está separada con un ";". Por ejemplo: "/coches; color = red; año = 2012". Para asignar varios valores a un atributo los valores se separan por una ","; por ejemplo; "color = rojo, verde, azul" o se repite el nombre de la variable "color = red; color = verde; color = blue".

Si se espera una URL que contiene las variables de matriz, el patrón de asignación solicitudes se debe representar con una plantilla URI. Esto asegura que la solicitud se pueda emparejar correctamente, independientemente de si las variables de matriz están presentes o no, y en qué orden se proporcionan.

A continuación se muestra un ejemplo de la extracción de la variable de matriz "q":

```
// GET /pets/42;q=11;r=22

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)
public void findPet(@PathVariable String petId, @MatrixVariable int q)
{

    // petId == 42
    // q == 11

}
```

Dado que todos los segmentos del path pueden contener variables de matriz es necesario especificar donde se encuentra la variable:

```
// GET /owners/42;q=11/pets/21;q=22

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method =
RequestMethod.GET)
public void findPet(
        @MatrixVariable(value="q", pathVar="ownerId") int q1,
        @MatrixVariable(value="q", pathVar="petId") int q2) {

    // q1 == 11
    // q2 == 22

}
```

Una variable de matriz puede ser definida como opcional y tener especificado un valor predeterminado:

```
// GET /pets/42

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)
public void findPet(@MatrixVariable(required=false, defaultValue="1")
int q) {

    // q == 1

}
```

Todas las variables de matriz se pueden obtener en un Map:

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method =
RequestMethod.GET)
public void findPet(
        @MatrixVariable Map<String, String> matrixVars,
        @MatrixVariable(pathVar="petId"") Map<String, String>
petMatrixVars) {

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 11, "s" : 23]
```

```
}
```

Hay que tener en cuenta que para poder usar las variables de matriz, se debe establecer la propiedad removeSemicolonContent como falso del RequestMappingHandlerMapping ya que por defecto se establece como verdadero.

Parámetros de la petición y Valores de la cabecera

Se puede reducir la asignación de peticiones mediante las condiciones en los parámetros de las peticiones como "myParam", "!myParam" o "myParam = myValue". A continuación se muestra un ejemplo:

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping(value = "/pets/{petId}", method =
RequestMethod.GET, params="myParam=myValue")
    public void findPet(@PathVariable String ownerId, @PathVariable
String petId, Model model) {
        // implementation omitted
    }

}
```

Se puede hacer lo mismo para la probar la presencia o la ausencia del encabezado de una solicitud o para comprobar la coincidencia de un valor específico con la cabecera de la solicitud:

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping(value = "/pets", method = RequestMethod.GET,
headers="myHeader=myValue")
    public void findPet(@PathVariable String ownerId, @PathVariable
String petId, Model model) {
        // implementation omitted
}}
```

3.3. Definición de un método controlador @RequestMapping

Un método controlador con la anotación @RequestMapping puede tener una estructura muy flexible. La mayoría de los argumentos se pueden utilizar en orden arbitrario, con la única excepción de BindingResult.

Tipos de argumentos soportados

Los siguientes son los argumentos admitidos por los métodos:

- Objetos de petición o respuesta (API Servlet). Es válido cualquier tipo específico de petición o de respuesta, por ejemplo ServletRequest o HttpServletRequest.
- Objeto Session (API Servlet), del tipo HttpSession. Este tipo de argumento establece la sesión correspondiente. Como consecuencia, este argumento no puede ser nulo.
- org.springframework.web.context.request.WebRequest u org.springframework.web.context.request.NativeWebRequest. Permite el acceso al parámetro de la petición, así como acceso a los atributos de petición o sesión, sin necesidad de vínculos con la API Servlet/portlets.
- java.util.Locale para obtener la configuración regional de la solicitud actual.
- java.io.InputStream/java.io.Reader para acceder al contenido de la solicitud.
- java.io.OutputStream / java.io.Writer para generar el contenido de la respuesta.
- org.springframework.http.HttpMethod para usar el método de la petición HTTP.
- java.security.Principal contiene al usuario autenticado.
- @PathVariable anotaciones de los parámetros de acceso a las variables de las plantillas URI.
- @MatrixVariable anotaciones de los parámetros para el acceso a las parejas nombre-valor ubicados en los segmentos del Path.
- @RequestParam anotaciones de los parámetros para el acceso a los parámetros específicos de la petición Servlet.
- @RequestHeader anotaciones de los parámetros para el acceso a las cabeceras específicas de la petición Servlet HTTP.
- @RequestBody anotaciones de los parámetros para el acceso al cuerpo de la solicitud HTTP.
- @RequestPart anotaciones de los parámetros para el acceso al contenido de una parte de la solicitud "multipart/form-data".
- HttpEntity <?> parámetros de acceso a las cabeceras HTTP de petición y contenidos del Servlet.
- java.util.Map/org.springframework.ui.Model/org.springframework.ui.ModelMap para enriquecer el modelo de la vista.
- org.springframework.web.servlet.mvc.support.RedirectAttributes para especificar el conjunto exacto de atributos a usar en caso de que se produzca una redirección y también para agregar atributos de flash.
- Comando o formulario para enlazar parámetros de la solicitud a las propiedades del bean, o directamente a los campos, con la conversión de tipos, dependiendo de los métodos @InitBinder y/o la configuración HandlerAdapter.
- org.springframework.validation.Errors/org.springframework.validation.BindingResult resultado de la validación de un comando o formulario anterior.

- org.springframework.web.bind.support.SessionStatus estado del controlador para marcar el proceso como completo, lo que desencadena la limpieza de los atributos de la sesión que han sido usados por la anotación @SessionAttributes.
- org.springframework.web.util.UriComponentsBuilder un constructor para la preparación de una dirección URL relativa al host de la solicitud actual, el puerto, el esquema, la ruta de contexto, y la parte literal del servlet.

Los parámetros Errors o BindingResult tienen que seguir el modelo de objetos que se ha especificado en la definición del método, sino, se podría tener más de un modelo de objetos y Spring crearía instancias de BindingResult para cada uno de ellos.

Mala utilización de BindingResult y @ModelAttribute.

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, Model
model, BindingResult result) { ... }
```

Hay que tener en cuenta, que hay un modelo de parámetros entre Pet y BindingResult. Para que esto funcione hay que reordenar los parámetros de la siguiente manera:

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet,
BindingResult result, Model model) { ... }
```

Tipos de devolución soportados

Los siguientes son los tipos de devolución soportados por los métodos:

- Un objeto ModelAndView, con el modelo implícitamente enriquecido, con objetos de comando y con los resultados de los métodos de acceso de datos de referencia con anotaciones @ModelAttribute.
- Un objeto Model, con el nombre de la vista implícita determinada a través de un RequestToViewNameTranslator y el modelo implícitamente enriquecido con objetos de comando y los resultados de los métodos de acceso de datos de referencia con anotaciones @ModelAttribute .
- Un objeto Map para la exposición de un modelo, con el nombre de la vista implícita determinada a través de un RequestToViewNameTranslator y el modelo implícitamente enriquecido con objetos de comando y los resultados de los métodos de acceso de datos de referencia con anotaciones @ModelAttribute .
- Un objeto View, con el modelo implícito determinado a través de un RequestToViewNameTranslator y el modelo implícitamente enriquecido con objetos de comando y los resultados de los métodos de acceso de datos de referencia con anotaciones @ModelAttribute. El método de control también se puede enriquecer mediante la declaración de un argumento Model.
- Un valor String que se interpreta como el nombre de vista lógico, con el modelo implícito determinado a través de un RequestToViewNameTranslator y el modelo implícitamente enriquecido con objetos de comando y los resultados de los métodos de

acceso de datos de referencia con anotaciones @ModelAttribute. El método de control también se puede enriquecer mediante la declaración de un argumento Model.

- No tendrá valor de retorno si el método es el encargado de escribir la respuesta o si se el nombre de la vista implícita se determina a través de una RequestToViewNameTranslator.
- Si el método tiene la anotación @ResponseBody, el tipo de retorno estará escrito en el cuerpo de la respuesta HTTP. El valor de retorno se convierte en el tipo de argumento declarado en el método HttpMessageConverter.
- Un objeto HttpEntity<?> o ResponseEntity<?> que facilitar el acceso a las cabeceras y contenidos HTTP de la respuesta del Servlet.
- Un objeto HttpHeaders que devuelve una respuesta sin cuerpo.
- Callable<?>, puede ser devuelto cuando la aplicación quiere producir el valor de retorno de forma asincrónica en un subproceso administrado por Spring MVC.
- DeferredResult<?> puede ser devuelto cuando la aplicación quiere producir el valor de retorno de un hilo de su propia elección.
- Cualquier otro tipo de retorno es considerado como un solo atributo del modelo que está expuesto a la vista, utilizando el nombre de atributo especificado a través de la anotación @ModelAttribute a nivel de método. El modelo se ha enriquecido de forma implícita con objetos de comando y con los resultados de los métodos de acceso de datos de referencia con anotaciones @ModelAttribute.

Unir parámetros de petición con parámetros de un método mediante @RequestParam

Para unir un parámetro de la petición a un parámetro de un método es necesario usar la anotación @RequestParam.

El siguiente fragmento de código muestra su uso:

```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(@RequestParam("petId") int petId, ModelMap
model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    } // ...}
```

Los parámetros que utilizan esta anotación son obligatorios, aunque se pueden establecer como opcionales poniendo el valor del atributo requiered como false (por ejemplo, @RequestParam (value="id", required=false)).

La conversión de tipos se aplica automáticamente si el tipo del método seleccionado no es String.

Mapeo del cuerpo de la petición con la anotación @ RequestBody

La anotación del parámetro de método @RequestBody indica que un parámetro del método debería regirse por el contenido del cuerpo de la petición HTTP. Por ejemplo:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws
IOException {
    writer.write(body);
}
```

@RequestBody convierte el cuerpo de la petición en un argumento del método utilizando el HttpMessageConverter. HttpMessageConverter es el responsable de convertir el mensaje de la petición HTTP en un objeto, y también, de convertir un objeto en el cuerpo de la respuesta HTTP. RequestMappingHandlerAdapter ayuda a la anotación @RequestBody a solventar los siguientes defectos del HttpMessageConverters:

- ByteArrayHttpMessageConverter convierte los arrays de bytes.
- StringHttpMessageConverter convierte los Strings.
- FormHttpMessageConverter convierte los datos del formulario en un MultiValueMap <String, String> o viceversa.

Para poder leer y escribir XML es necesario configurar el MarshallingHttpMessageConverter con una implementación específica de Marshaller y de Unmarshaller del paquete org.springframework.oxm como en el siguiente ejemplo:

```
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMa
ppingHandlerAdapter">
    <property name="messageConverters">
        <util:list id="beanList">
            <ref bean="stringHttpMessageConverter"/>
            <ref bean="marshallingHttpMessageConverter"/>
        </util:list>
    </property>
</bean>

<bean id="stringHttpMessageConverter"

class="org.springframework.http.converter.StringHttpMessageConverter"/
>

<bean id="marshallingHttpMessageConverter"

class="org.springframework.http.converter.xml.MarshallingHttpMessageCo
nverter">
    <property name="marshaller" ref="castorMarshaller" />
    <property name="unmarshaller" ref="castorMarshaller" />
</bean>

<bean id="castorMarshaller"
class="org.springframework.oxm.castor.CastorMarshaller"/>
```

Un parámetro de un método @RequestBody puede ser anotado con @Valid, en cuyo caso, será validado usando la configuración de la instancia Validator. Cuando se utiliza el espacio de nombres del MVC o la config MVC Java, un validador JSR-303 se configura automáticamente asumiendo una implementación JSR-303 que se encuentre disponible en la ruta de clases.

Al igual que con los parámetros @ModelAttribute, el argumento Errors puede ser utilizado para examinar los errores. Si no se declara tal argumento, un método MethodArgumentNotValidException será ejecutado. La excepción se manejara desde el DefaultHandlerExceptionResolver, que enviará un error 400 de vuelta al cliente.

Mapeo del cuerpo de la respuesta con la anotación @ResponseBody

La anotación @ResponseBody es similar a @RequestBody. Esta anotación se puede poner en un método e indicar el tipo de retorno debe producirse en el cuerpo de la respuesta HTTP. Por ejemplo:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
@ResponseBody
public String helloWorld() {
    return "Hello World";
}
```

El ejemplo anterior el texto Hello World se escribirá en la secuencia de respuesta HTTP.

Al igual que con @RequestBody, Spring convierte el objeto devuelto en el cuerpo de la respuesta mediante el uso de un HttpMessageConverter.

Crear controladores REST con la anotación @RestController

Es un caso de uso muy común tener controladores implementan una API REST, para los que solo sirven contenido JSON, XML o contenido personalizado MediaType. Para mayor comodidad, Spring permite poner la anotación @RestController en la clase Controlado en lugar de hacerlo en todos los métodos @RequestMapping.

@RestController es una anotación de estereotipo que combina @ResponseBody y @Controller.

HttpEntity

HttpEntity es similar a @RequestBody y @ResponseBody. Además de obtener acceso al cuerpo de la petición y de la respuesta, HttpEntity permite el acceso a las cabeceras de petición y respuesta, de esta forma:

```
@RequestMapping("/something")
public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity)
throws UnsupportedEncodingException {
    String requestHeader =
requestEntity.getHeaders().getFirst("MyRequestHeader"));
    byte[] requestBody = requestEntity.getBody();
```

```
    // do something with request header and body

    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("MyResponseHeader", "MyValue");
    return new ResponseEntity<String>("Hello World", responseHeaders,
HttpStatus.CREATED);
}
```

En el ejemplo anterior se obtiene el valor del encabezado MyRequestHeader de la solicitud, y se lee el cuerpo como una matriz de bytes, además se agrega MyResponseHeader a la respuesta, se escribe Hello World en la secuencia de respuesta y se establece el código del estado de respuesta como 201.

Al igual que con @RequestBody y @ResponseBody, Spring utiliza HttpMessageConverter para convertir los segmentos de solicitud y respuesta.

@ModelAttribute en un método

La anotación @ModelAttribute se puede utilizar a nivel de método o a nivel de argumento. En esta sección se explica su uso en los métodos, mientras que la siguiente sección se explica su uso en los argumentos del método.

@ModelAttribute en un método indica que el propósito de ese método consiste en agregar uno o más atributos al modelo. Estos métodos son compatibles con los mismos tipos de argumentos que los métodos @RequestMapping, pero no se pueden asignar directamente a las solicitudes. Un controlador puede tener cualquier número de métodos @ModelAttribute, y estos son invocados antes que los métodos @RequestMapping del mismo controlador.

A continuación se presentan dos estilos de utilización de esta anotación:

```
// Add one attribute
// The return value of the method is added to the model under the name
"account"
// You can customize the name via @ModelAttribute("myAccount")

@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountManager.findAccount(number);
}

// Add multiple attributes

@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountManager.findAccount(number));
    // add more ...
}
```

En el primer estilo, el método añade un atributo implícitamente en el return, y en el segundo, el método acepta un modelo y le agrega cualquier número de atributos.

Un método @ModelAttribute también puede ser definido en una clase @ControllerAdvice en la que sus métodos se apliquen a varios controladores.

La anotación @ModelAttribute a se puede utilizar también en métodos @RequestMapping para que el valor de retorno del método se interprete como un atributo del modelo y no como un nombre de vista.

@ModelAttribute en un argumento de método

Como se ha explicado en la sección anterior la anotación @ModelAttribute se puede utilizar a nivel de método o a nivel de argumento. En esta sección se explica su uso en los argumentos del método.

@ModelAttribute en un argumento indica que este debe ser recuperado a partir del modelo. Si el argumento no está presente en el modelo, el argumento debe ser instanciado y añadido al modelo. Una vez presente en el modelo, los campos del argumento deben ser rellenados con todos los parámetros que coincidan con los de la petición. Esto se conoce en Spring MVC como enlace de datos, y es un mecanismo muy útil que ahorra tener que analizar cada campo del formulario de forma individual.

Un método @ModelAttribute es una forma de recuperar un atributo de la base de datos, que puede estar almacenado entre las peticiones mediante el uso de @SessionAttributes. En algunos casos para recuperar el atributo puede ser conveniente usar una variable de plantilla URI y un convertidor de tipos. He aquí un ejemplo:

```
@RequestMapping(value="/accounts/{account}", method =
RequestMethod.PUT)
public String save(@ModelAttribute("account") Account account) {

}
```

En este ejemplo, el nombre del atributo del modelo (account) coincide con el nombre de una variable de plantilla URI.

@SessionAttributes para definir atributos de sesión

La anotación @SessionAttributes declara atributos de sesión utilizados por un controlador específico. Normalmente, mostrará una lista de los atributos del que deben almacenarse de forma transparente en la sesión.

El siguiente fragmento de código muestra el uso de esta anotación, especificando el atributo del modelo:

```
@Controller
@RequestMapping("/editPet.do")
@SessionAttributes("pet")
public class EditPetForm {
    // ...
}
```

La anotación @CookieValue

La anotación @CookieValue obliga al parámetro de un método a tomar el valor de una cookie HTTP.

Consideremos que la siguiente cookie ha sido recibida con una petición http:

Jsessionid = 415A4AC178C59DACE0B2C9CA727CDD84

El siguiente ejemplo muestra cómo obtener el valor de la cookie JSESSIONID:

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@CookieValue("JSESSIONID") String
cookie) {//...}
```

La conversión de tipos se realiza automáticamente si el tipo de parámetro del método objetivo no es un String. Esta anotación es compatible con las anotaciones de en entornos Servlet y Portlets.

@RequestHeader y los atributos de las cabeceras de peticiones

La anotación @RequestHeader permite enlazar un parámetro de un método a la cabecera de una solicitud.

Aquí se muestra es una cabecera de una solicitud:

Host localhost: 8080
Acepte text / html, application / xhtml + xml, application / xml; q = 0,9
Accept-Language fr, es-es, q = 0,7, es; q = 0,3
Gzip Accept-Encoding, desinfle
Accept-Charset ISO-8859-1, UTF-8, q = 0,7, *; q = 0,7
Keep-Alive 300

El siguiente ejemplo de código muestra cómo obtener el valor de los atributos Accept-Encoding y Keep-Alive que se encuentran dentro de la cabecera:

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String
encoding,
        @RequestHeader("Keep-Alive") long keepAlive) {
    //...
}
```

La conversión de tipos se realiza automáticamente si el tipo de parámetro del método objetivo no es un String. Esta anotación es compatible con las anotaciones de en entornos Servlet y Portlets.

Parámetros del método y la conversión de tipos

Los valores basados en Strings extraídos de la petición incluyendo, parámetros de la petición, variables path, los encabezados de solicitud y los valores de cookie, pueden necesitar ser

convertidos al tipo del parámetro del método. Si el tipo de destino no es String, Spring lo convierte automáticamente al tipo necesario. Se admiten todos los tipos simples como int, long, fecha, etc. También se puede personalizar el proceso de conversión a través de WebDataBinder registrando Formatters con el FormattingConversionService.

Personalización de WebDataBinder

Para personalizar los parámetro de la petición vinculada a PropertyEditors mediante Spring WebDataBinder se usa la anotación para métodos @InitBinder dentro del controlador, dentro de una clase @ControllerAdvice o a través de un WebBindingInitializer personalizado.

Personalización del enlace de datos con @InitBinder

La anotación @InitBinder, para métodos de controlador, permite configurar los datos de Web vinculados directamente con la clase del controlador. @InitBinder identifica los métodos que inicializa el WebDataBinder, que se utilizará para introducir objetos en forma de comandos y formularios de los métodos anotados en el controlador.

Estos métodos Init soportan todos los argumentos que @RequestMapping pueda tener, excepto los objetos comando/formulario y los correspondientes objetos de validación. Los métodos Init para enlace de datos no tienen valor de retorno, por lo tanto, generalmente se declaran como void.

El siguiente ejemplo muestra el uso de @InitBinder para configurar un CustomDateEditor para todos los java.util.Date de los formularios.

```
@Controller
public class MyFormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new
CustomDateEditor(dateFormat, false));
    }

    // ...}
```

Configuración personalizada de un WebBindingInitializer

Para exteriorizar la inicialización del enlace de datos, se puede proporcionar una implementación personalizada de la interfaz WebBindingInitializer, que luego se habilitará mediante la configuración personalizada de un bean con AnnotationMethodHandlerAdapter, para anular la configuración por defecto.

El siguiente ejemplo muestra una configuración usando una implementación personalizada de la interfaz WebBindingInitializer, org.springframework.samples.petclinic.web.ClinicBindingInitializer, que configura el PropertyEditors requerido por varios controladores.

```
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMa
ppingHandlerAdapter">
    <property name="cacheSeconds" value="0" />
    <property name="webBindingInitializer">
        <bean
class="org.springframework.samples.petclinic.web.ClinicBindingInitiali
zer" />
    </property>
</bean>
```

3.4 Peticiones asíncronas

Spring MVC 3.2 introdujo Servlet 3 basado en el procesamiento de solicitudes asincrónicas. En lugar de devolver un valor, un método controlador puede devolver un java.util.concurrent.Callable y producir el valor de retorno de un hilo distinto. Mientras tanto el hilo principal del Servlet se libera y permite procesar otras solicitudes. Spring MVC invoca el Callable en un hilo separado con la ayuda de un TaskExecutor y cuando el Callable finaliza, la solicitud se envía de vuelta al contenedor de Servlets para reanudar el procesamiento con el valor devuelto por el Callable. He aquí un ejemplo de un método controlador:

```
@RequestMapping(method=RequestMethod.POST)
public Callable<String> processUpload(final MultipartFile file) {

    return new Callable<String>() {
        public String call() throws Exception {
            // ...
            return "someView";
        }
    };

}
```

Otra opción es que el controlador devuelva una instancia de DeferredResult. En este caso, el valor de retorno también se produce a partir de un hilo secundario. Sin embargo, ese hilo no es conocido por Spring MVC. Por ejemplo, el resultado puede ser producido en respuesta a algún evento externo, como un mensaje de JMS, una tarea programada, etc. Aquí se muestra un ejemplo de un método controlador:

```
@RequestMapping("/quotes")
@ResponseBody
public DeferredResult<String> quotes() {
    DeferredResult<String> deferredResult = new
DeferredResult<String>();
    // Save the deferredResult in in-memory queue ...
    return deferredResult;
}

// In some other thread...
deferredResult.setResult(data);

/ / En otro hilo ...
```

```
deferredResult.setResult (datos);
```

La siguiente es la secuencia de eventos para el procesamiento asíncrono de una solicitud con un método Callable:

1. El controlador devuelve un Callable.
2. Spring MVC inicia el procesamiento asincrónico y envía el Callable a un TaskExecutor para procesarlo en un hilo separado.
3. El DispatcherServlet y todos los filtros cierran el hilo del procesamiento de solicitudes, pero la respuesta sigue abierta.
4. El Callable produce un resultado y Spring MVC envía la petición de vuelta al contenedor de Servlets.
5. El DispatcherServlet se invoca de nuevo y el procesamiento se reanuda con el resultado producido de forma asíncrona desde el Callable.

La secuencia de los pasos 2, 3, y 4 puede variar dependiendo de la velocidad de ejecución de los hilos concurrentes.

La secuencia de eventos de procesamiento asíncrono de una solicitud con un DeferredResult es en principio igual, salvo que depende de la aplicación para obtener el resultado asincrónico de un hilo:

1. El controlador devuelve un DeferredResult y lo guarda en alguna cola o lista en memoria.
2. Spring MVC inicia el procesamiento asincrónico.
3. El DispatcherServlet todos los filtros cierran el hilo del procesamiento de solicitudes, pero la respuesta sigue abierta.
4. La aplicación establece el DeferredResult del hilo y Spring MVC envía la solicitud al contenedor de Servlets.
5. El DispatcherServlet se invoca de nuevo y el procesamiento se reanuda con el resultado producido de forma asíncrona.

Manejo de excepciones para las peticiones asíncronas

¿Qué sucede si un return de un método Callable de un controlador genera una excepción mientras se está ejecutando? El efecto es similar a lo que ocurre cuando un método controlador genera una excepción. Es tratada por un método @ExceptionHandler con el mismo controlador o por una de las instancias HandlerExceptionResolver.

Cuando se utiliza un DeferredResult, se tiene la opción de llamar a un método setErrorResult (Object) y proporcionar una excepción o cualquier otro objeto que se quiera obtener como resultado. Si el resultado es una excepción, se procesa con un método @ExceptionHandler en el mismo controlador o con cualquier instancia HandlerExceptionResolver.

Una opción para volver a iniciar el ciclo de vida de una petición asincrónica proviene directamente de DeferredResult, que tiene los métodos OnTimeOut (Runnable) y onCompletion (Runnable). Se llaman cuando la solicitud asincrónica está a punto de agotar el tiempo de vida o se ha completado, respectivamente. El evento OnTimeOut puede ser

controlado por la asignación de DeferredResult con algún valor. Sin embargo onCompletion es final y el resultado ya no se puede cambiar.

Intercepción de solicitudes asíncronas

HandlerInterceptor puede implementar AsyncHandlerInterceptor, que proporciona un método adicional afterConcurrentHandlingStarted. HandlerInterceptor se invoca después de iniciar el procesamiento asincrónico y cuando el hilo de procesamiento de la solicitud inicial se cierra.

Configuración para el Procesamiento de solicitudes asíncronas

Servlet 3 Async Config

Para utilizar el procesamiento de la solicitud Servlet 3 async, es necesario actualizar web.xml a la versión 3.0:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    ...

</web-app>
```

El DispatcherServlet y cualquier filtro de configuración necesita tener el sub-elemento <async-supported>true</async-supported>. Además, cualquier filtro tiene que involucrarse en envíos asincrónicos, por lo que debe ser configurado para soportar el tipo envió ASYNC.

Spring MVC Async Config

El config MVC de Java y el espacio de nombres MVC proporcionan opciones para configurar el proceso de peticiones asíncronas. WebMvcConfigurer tiene el método configureAsyncSupport mientras <mvc: annotation-driven> tiene un sub-elemento <async-support>.

Estos permiten configurar el valor de tiempo de espera predeterminado para las peticiones asíncronas, que si no se establece depende del contenedor de Servlets subyacente (por ejemplo, 10 segundos en Tomcat). También se puede configurar un AsyncTaskExecutor para ejecutar instancias Callable devueltas desde los métodos del controlador. Es muy recomendable configurar esta propiedad ya que por defecto Spring MVC utiliza SimpleAsyncTaskExecutor. El config MVC Java y el espacio de nombres MVC también le permiten registrar instancias CallableProcessingInterceptor y DeferredResultProcessingInterceptor.

Si se necesita reemplazar el valor de tiempo de espera predeterminado para un DeferredResult, puede hacerse utilizando el constructor de la clase apropiada. Del mismo modo un Callable puede envolverse en una WebAsyncTask y utilizar el constructor de la clase apropiada para

personalizar el valor de tiempo de espera. El constructor de la clase de WebAsyncTask también permite proporcionar una AsyncTaskExecutor.

4. Mapeo de controladores

Con la introducción de los controladores anotados, el RequestMappingHandlerMapping busca automáticamente anotaciones @RequestMapping en todos los beans @Controller. Sin embargo, hay que tener en cuenta que todas las clases HandlerMapping extienden de AbstractHandlerMapping y tienen las siguientes propiedades:

- Interceptors. Lista de interceptores para su uso.
- DefaultHandler utilizado por defecto, cuando la asignación de controlador no encuentra ningún controlador.
- Order. Basado en el valor de la propiedad de Order, Spring compara todas las asignaciones de control disponibles y aplica el primer controlador coincidente.
- AlwaysUseFullPath Si su valor es true, Spring utiliza la ruta completa en el contexto actual para encontrar un controlador adecuado. Si es false (por defecto), se utiliza la ruta de acceso dentro de la correlación del Servlet actual.
- UrlDecode El valor predeterminado es true. Para comparar las rutas codificadas, su valor debe ser false.

El siguiente ejemplo muestra cómo configurar un interceptor:

```
<beans>
    <bean id="handlerMapping"
class="org.springframework.web.servlet.mvc.method.annotation.RequestMa
ppingHandlerMapping">
        <property name="interceptors">
            <bean class="example.MyInterceptor"/>
        </property>
    </bean>
<beans>
```

4.1 La interceptación de peticiones con un HandlerInterceptor

El mecanismo de mapeo de controladores de Spring incluye interceptores de controladores, que son útiles cuando se desea aplicar una funcionalidad específica a ciertas peticiones. La clase HandlerInterceptorAdapter hace que sea más fácil de extender la interfaz HandlerInterceptor.

Los interceptores ubicados en el mapeo de controladores deben implementar los HandlerInterceptor del paquete org.springframework.web.servlet. Esta interfaz define tres métodos: preHandle(..) que se llama antes de que se ejecute el controlador real; postHandle(..) que se llama después de que se ejecute el controlador y afterCompletion (..) que se llama después de que la solicitud completa haya finalizado. Estos tres métodos deben proporcionar la flexibilidad suficiente para hacer todo tipo de pre-procesamiento y post-procesamiento.

El método preHandle(..) devuelve un valor booleano. Este método se puede utilizar para terminar o continuar con el procesamiento de la cadena de ejecución. Cuando este método devuelve verdadero, la cadena de ejecución del gestor continuará; cuando se devuelve false, el DispatcherServlet asume que el propio interceptor se ha encargado de las peticiones y detiene la ejecución de los otros interceptores y de la cadena de ejecución del controlador real.

Los interceptores se pueden configurar mediante la propiedad Interceptors, que está presente en todas las clases HandlerMapping que extienden de AbstractHandlerMapping. Esto se muestra en el ejemplo siguiente:

```
package samples;

public class TimeBasedAccessInterceptor extends
HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;

    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }

    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }

    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response,
            Object handler) throws Exception {
        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        if (openingTime <= hour && hour < closingTime) {
            return true;
        }

response.sendRedirect("http://host.com/outsideOfficeHours.html");
        return false;
    }
}
```

Cualquier solicitud de control mediante este mapeo es interceptado por TimeBasedAccessInterceptor. En el ejemplo anterior si la hora actual está fuera del horario acotado, el usuario es redirigido a un archivo HTML estático.

5. Resolver vistas

Todos los frameworks MVC para aplicaciones Web proporcionan una forma para hacer frente a puntos de vistas. Spring proporciona resolución de vistas, que le permite hacer modelos en un navegador sin necesidad de vincularlos a una tecnología de vistas específica. Spring permite utilizar JSP, plantillas Velocity y vistas XSLT, entre otras.

Las dos interfaces más importantes con las que Spring maneja las vistas son ViewResolver y View. ViewResolver proporciona una correspondencia entre los nombres de vista y las vistas reales. La interfaz View prepara la solicitud y pasa la petición a una de las tecnologías de vistas.

5.1 La interfaz ViewResolver

Todos los métodos del controlador de Spring Web MVC controlan como resolver un nombre de vista lógico, ya sea de forma explícita o implícita. Las vistas en Spring son abordadas con un nombre de vista lógico y se resuelven mediante una resolución de la vista. Esta tabla muestra la mayoría de ellos, seguidos de un par de ejemplos.

ViewResolver	Descripción
AbstractCachingViewResolver	A menudo las vistas necesitan preparación antes de ser utilizadas; la extensión de este ViewResolver proporciona el almacenamiento en caché de la vista.
XmlViewResolver	Acepta un archivo de configuración escrito en XML con el mismo DTD que las fábricas de beans XML de Spring. El archivo de configuración por defecto es /WEB-INF/views.xml.
ResourceBundleViewResolver	Utiliza definiciones de beans en un ResourceBundle. Normalmente, se define el paquete en un archivo de propiedades, que se encuentra en la ruta de clases. El nombre de archivo predeterminado es views.properties.
UrlBasedViewResolver	Efectúa la resolución directa del nombre lógico de las vistas como direcciones URL sin una definición de asignación explícita.
InternalResourceViewResolver	Subclase de UrlBasedViewResolver que soporta InternalResourceView y subclases como JstlView y TilesView. Se puede especificar la clase de vista para todas las vistas generadas por esta resolución mediante el uso de setViewClass(..)
VelocityViewResolver / FreeMarkerViewResolver	Subclase de UrlBasedViewResolver que soporta VelocityView, oFreeMarkerView y sus subclases personalizadas.
ContentNegotiatingViewResolver	Resuelve una vista basada en el nombre del archivo o del controlador de la petición Accept.

Figura 25. ViewResolvers de Spring.

A modo de ejemplo, con JSP como tecnología de vista, se puede utilizar el UrlBasedViewResolver. Esta resolución de vista traduce un nombre de vista a una URL y pasa la petición a la RequestDispatcher para representar la vista.

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

Cuando se combinan diferentes tecnologías de vista en una aplicación web, se puede utilizar el ResourceBundleViewResolver:

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.ResourceBundleViewResolver
">
    <property name="basename" value="views"/>
    <property name="defaultParentView" value="parentView"/>
</bean>
```

ResourceBundleViewResolver inspecciona el ResourceBundle identificado por el nombre base cada vista que debe resolver, se utiliza el valor de la propiedad [viewname].(class) como la clase de vista y el valor de la propiedad [viewname].url como la URL de la vista.

5.2 Encadenar las resoluciones de vistas

Spring admite varias resoluciones de la vista. De esta manera se pueden resolver varias vistas en cadena y, por ejemplo, anular vistas específicas en determinadas circunstancias. Se pueden resolver vistas en cadena añadiendo más de una resolución al contexto de aplicación.

El siguiente ejemplo está compuesto por dos resolvers, un InternalResourceViewResolver, que siempre se coloca automáticamente como la última resolución de la cadena, y un XmlViewResolver para especificar vistas en Excel. Las vistas de Excel no son compatibles con el InternalResourceViewResolver.

```
<bean id="jspViewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolv
er">
    <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>

<bean id="excelViewResolver"
class="org.springframework.web.servlet.view.XmlViewResolver">
    <property name="order" value="1"/>
    <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in views.xml -->

<beans>
    <bean name="report"
class="org.springframework.example.ReportExcelView"/>
</beans>
```

Si una resolución de vista específica no tiene como resultado una vista, Spring examina el contexto de otras resoluciones de vista. Si existen resoluciones de vista adicionales, Spring sigue inspeccionarlos hasta que se resuelva una vista. Si no hay resoluciones de vistas que devuelvan una vista, Spring lanza un ServletException.

Una resolución de la vista puede devolver null para indicar que la vista no se pudo encontrar, pero no todas las resoluciones de vista lo hacen, sin embargo, en algunos casos, la resolución simplemente no puede detectar si existe o no la vista.

5.3 Redireccionamiento de vistas

Un controlador normalmente devuelve un nombre de vista lógico que una resolución de vista convierte a una tecnología de vista particular. Para tecnologías de vistas como JSP que se procesan a través del Servlet o del motor JSP, esta resolución suele ser manejada a través de la combinación de InternalResourceViewResolver y InternalResourceView, que usa el método RequestDispatcher.forward(..) de la API Servlet o el método RequestDispatcher.include(). Para otras tecnologías de vista, tales como la Velocity, XSLT, etc., la vista en sí escribe su contenido directamente en la secuencia de respuesta.

RedirectView

Una forma de forzar una redirección como el resultado de una respuesta del controlador es crear y devolver una instancia de RedirectView. En este caso, DispatcherServlet no utiliza el mecanismo normal de resolución de vistas. Por el contrario, ya que se ha redirigido la vista el DispatcherServlet simplemente instruye a la vista para hacer su trabajo.

RedirectView emite una llamada a HttpServletResponse.sendRedirect() que devuelve al navegador del cliente una redirección HTTP. Por defecto todos los atributos del modelo son considerados como variables de la plantilla URI en la URL de redirección. De los atributos restantes, los que son tipos primitivos o colecciones/arrays de tipos primitivos se adjuntan automáticamente como parámetros de consulta.

Anexar atributos de tipo primitivo como parámetros de consulta puede ser beneficioso si una instancia del modelo está específicamente preparada para la redirección. Sin embargo, los controladores del modelo pueden contener atributos adicionales añadidos para fines de representación. Para evitar la posibilidad de que tales atributos aparezcan en la URL un controlador puede declarar un argumento como tipo RedirectAttributes y utilizarlo para especificar los atributos exactos que se ponen a disposición de RedirectView. Si el método del controlador decide hacer la redirección, se utiliza el contenido de RedirectAttributes, de lo contrario, se utiliza el contenido del modelo.

Algo a tener en cuenta es que las variables de la plantilla URI de las peticiones se hacen automáticamente cuando se expande una URL de redirección, por lo que no es necesario añadirlas explícitamente ni mediante Model ni por RedirectAttributes. Por ejemplo:

```
@RequestMapping(value = "/files/{path}", method = RequestMethod.POST)
public String upload(...) {
```

```
    // ...
    return "redirect:files/{path}";
}
```

5.4 ContentNegotiatingViewResolver

ContentNegotiatingViewResolver no resuelve vistas por sí mismo, sino que delega esta función a otros resolvers, seleccionando la vista que más se asemeja a la representación solicitada por el cliente. Existen dos estrategias para solicitar una representación del servidor:

- Utilizar un URI distintivo para cada recurso, por lo general mediante el uso de una extensión de archivo diferente en el URI.
- Utilizar el mismo URI que el cliente para localizar el recurso, pero estableciendo la aceptación del encabezado de la solicitud HTTP para enumerar los tipos media.

Para soportar múltiples representaciones de un recurso, Spring ofrece el ContentNegotiatingViewResolver para resolver una vista basada en la extensión de un archivo o aceptar el encabezado de una solicitud HTTP. ContentNegotiatingViewResolver no realiza la resolución de la vista, pero en vez eso delega a una lista de resolvers que se especifica a través de la propiedad ViewResolvers de los beans.

ContentNegotiatingViewResolver selecciona una vista apropiada para atender la solicitud comparando el tipo media de la petición con el tipo media soportado por la vista asociada a cada uno de sus ViewResolvers. La primera vista de la lista que tiene un Content-Type compatible devuelve la representación del cliente. Si una vista compatible no puede ser suministrada por la cadena de ViewResolvers, entonces se consulta la lista de vistas específicas a través de la propiedad DefaultViews.

He aquí un ejemplo de configuración de un ContentNegotiatingViewResolver:

```
<bean
class="org.springframework.web.servlet.view.ContentNegotiatingViewReso
lver">
    <property name="mediaTypes">
        <map>
            <entry key="atom" value="application/atom+xml"/>
            <entry key="html" value="text/html"/>
            <entry key="json" value="application/json"/>
        </map>
    </property>
    <property name="viewResolvers">
        <list>
            <bean
class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
            <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolv
er">
                <property name="prefix" value="/WEB-INF/jsp/"/>
                <property name="suffix" value=".jsp"/>
            </bean>
        </list>
```

```
    </property>
    <property name="defaultViews">
        <list>
            <bean
class="org.springframework.web.servlet.view.json.MappingJackson2JsonVi
ew" />
        </list>
    </property>
</bean>

<bean id="content"
class="com.springsource.samples.rest.SampleContentAtomView"/>
```

El InternalResourceViewResolver se encarga de la traducción de los nombres de vista y las páginas JSP, mientras que el BeanNameViewResolver devuelve una vista basada en el nombre de un bean. En este ejemplo, el content del bean es una clase que hereda de AbstractAtomFeedVie, que devuelve un Atom RSS.

6. Construcción URI

Spring MVC proporciona un mecanismo para la construcción y la codificación de un URI utilizando UriComponentsBuilder y UriComponents.

Por ejemplo, para ampliar y codificar una cadena de plantilla URI:

```
UriComponents uriComponents = UriComponentsBuilder.fromUriString(

"http://example.com/hotels/{hotel}/bookings/{booking}").build();

URI uri = uriComponents.expand("42", "21").encode().toUri();
```

Hay que tener en cuenta que UriComponents es inmutable y las operaciones expand() y encode() devuelven nuevas instancias si es necesario.

También se puede ampliar y codificar utilizando componentes individuales URI:

```
UriComponents uriComponents = UriComponentsBuilder.newInstance()

.scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{bo
oking}").build()
        .expand("42", "21")
        .encode();
```

En un entorno Servlet la sub-clase ServletUriComponentsBuilder proporciona métodos generadores estáticos para copiar la información URL disponible desde las peticiones Servlet:

```
HttpServletRequest request = ...
```

```
// Re-use host, scheme, port, path and query string
// Replace the "accountId" query param

ServletUriComponentsBuilder ucb =
ServletUriComponentsBuilder.fromRequest(request)
        .replaceQueryParam("accountId", "{id}").build()
        .expand("123")
        .encode();
```

Alternativamente, se puede optar por copiar un subconjunto de los datos disponibles incluyendo la ruta de contexto:

```
// Re-use host, port and context path
// Append "/accounts" to the path

ServletUriComponentsBuilder ucb =
ServletUriComponentsBuilder.fromContextPath(request)
        .path("/accounts").build()
```

O en caso de que el DispatcherServlet este mapeado por su nombre (por ejemplo, /main/*), también se puede tener incluido la parte literal en el mapeo del Servlet:

```
// Re-use host, port, context path
// Append the literal part of the servlet mapping to the path
// Append "/accounts" to the path

ServletUriComponentsBuilder ucb =
ServletUriComponentsBuilder.fromServletMapping(request)
        .path("/accounts").build()
```

7. Construcción URI con controladores y métodos

Spring MVC proporciona otro mecanismo para la construcción y la codificación URI que enlazan con los controladores y los métodos definidos dentro de una aplicación. MvcUriComponentsBuilder extiende UriComponentsBuilder y ofrece tales posibilidades.

Dado este controlador:

```
@Controller
@RequestMapping("/hotels/{hotel}")
public class BookingController {

    @RequestMapping("/bookings/{booking}")
    public String getBooking(@PathVariable Long booking) {

    // ...
}}
```

Y el usando MvcUriComponentsBuilder, el ejemplo anterior quedaría:

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodName(BookingController.class,
"getBooking",21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

MvcUriComponentsBuilder también puede crear "Controladores de simulacros", lo que permite crear URI codificando contra el API actual del Controlador:

```
UriComponents uriComponents = MvcUriComponentsBuilder

.fromMethodCall(on(BookingController.class).getBooking(21)).buildAndEx
pand(42);

URI uri = uriComponents.encode).toUri();
```

8. Uso de la configuración regional

Spring Framework Web MVC soporta la internacionalización. DispatcherServlet permite resolver automáticamente los mensajes utilizando la configuración regional del cliente mediante objetos LocaleResolver.

Cuando llega una petición, el DispatcherServlet busca un LocaleResolver, si encuentra uno trata de usarlo para establecer la configuración regional de la petición. Usando el método RequestContext.getLocale () se puede obtener la configuración regional que se obtuvo por medio del LocaleResolver.

Además se puede añadir un interceptor para realizar el mapeo del controlador, para cambiar la configuración regional en determinadas circunstancias, por ejemplo, dependiendo de un parámetro en la solicitud.

Los LocaleResolvers y los interceptores se definen en el paquete org.springframework.web.servlet.i18n y se configuran en el contexto de aplicación. En las siguientes secciones se veran los tipos de resolutores de configuración regional que incluye Spring MVC.

8.1. Obtención de información de la zona horaria

Además de obtener la configuración regional del cliente, a menudo es útil conocer su zona horaria. La interfaz LocaleContextResolver ofrece una extensión LocaleResolver que permite proporcionar un LocaleContext más detallado, que puede incluir información de zona horaria.

Cuando está disponible, la zona horaria se puede obtener utilizando el método RequestContext.getTimeZone (). La información de zona horaria se utilizará automáticamente para convertir y dar formato a objetos registrados en Spring ConversionService.

8.2. AcceptHeaderLocaleResolver

Esta resolución de la configuración regional inspecciona la cabecera accept-language de la solicitud enviada por el cliente. Por lo general, este campo de la cabecera contiene la configuración regional del sistema operativo del cliente. Esta resolución no admite la información de la zona horaria.

8.3. CookieLocaleResolver

Esta resolución de la configuración regional inspecciona un Cookie que pudiera existir en el cliente para comprobar si se ha especificado una configuración regional o zona horaria. Si es así, utiliza los datos especificados, usando las propiedades de la resolución de configuración regional, se puede especificar el nombre de la cookie, así como su edad máxima. A continuación se muestra un ejemplo de la definición de un CookieLocaleResolver seguido de las propiedades de un CookieLocaleResolver.

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- in seconds. If set to -1, the cookie is not persisted
(deleted when browser shuts down) -->
    <property name="cookieMaxAge" value="100000">

</bean>
```

Propiedad	Defecto	Descripción
cookieName	classname + LOCALE	El nombre de la cookie
cookieMaxAge	Integer.MAX_INT	El tiempo máximo que una cookie persistente permanece en el cliente. Si se especifica -1, la cookie no persistirá, sino que sólo estará disponible hasta el cliente cierre el navegador.
cookiePath	/	Cuando se especifica cookiePath, la cookie sólo será

		visible en esa ruta y en las rutas por debajo de ella.

Figura 26. Propiedades de CookieLocaleResolver.

8.4. SessionLocaleResolver

SessionLocaleResolver permite recuperar la configuración regional y de la zona horaria de la sesión que podría estar asociada con la petición del usuario.

8.5. LocaleChangeInterceptor

Se puede habilitar la modificación de las configuraciones regionales mediante la adición del LocaleChangeInterceptor al mapeo del controlador. Se detectará un parámetro en la petición y cambiara la configuración regional. El siguiente ejemplo muestra que las llamadas a todos los recursos *.views contienen un parámetro llamado siteLanguage que cambia la configuración regional. Así, por ejemplo, la solicitud de la siguiente URL, http://www.sf.net/home.view?siteLanguage=nl cambiará el idioma del sitio al holandés.

```
<bean id="localeChangeInterceptor"

class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver"

class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping
">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/*/.view=someController</value>
    </property>
</bean>
```

9. Uso de temas

9.1. Información general sobre temas

Se pueden aplicar temas a Spring MVC para configurar el aspecto de la aplicación, mejorando así la experiencia del usuario. Un tema es una colección de recursos estáticos que afectan el estilo visual de la aplicación, por lo general el estilo del fondo y de las imágenes.

9.2. Definición de temas

Para utilizar los temas en la aplicación web, primero se debe configurar una implementación de la interfaz org.springframework.ui.context.ThemeSource. La interfaz WebApplicationContext extiende ThemeSource pero delega su responsabilidad en una implementación dedicada. Por defecto, el delegado será una implementación org.springframework.ui.context.support.ResourceBundleThemeSource que carga los archivos de propiedades desde la raíz de la ruta de clases. Para utilizar una implementación personalizada de ThemeSource o para configurar el nombre base del prefijo de ResourceBundleThemeSource, se registra un bean en el contexto de la aplicación con el nombre reservado themeSource. El contexto de aplicación Web detectará automáticamente el bean con ese nombre y lo utilizará.

Cuando se utiliza el ResourceBundleThemeSource, un tema se define en un archivo de propiedades y este enumera los recursos que componen el tema. He aquí un ejemplo:

styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg

Las llaves de las propiedades son los nombres que se refieren a los elementos temáticos del código de la vista. Para un JSP, por lo general se usa etiquetas personalizadas spring:theme, que son muy similares a las etiquetas spring:message. En el siguiente fragmento JSP se utiliza el tema definido en el ejemplo anterior para personalizar la apariencia:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
    <head>
        <link rel="stylesheet" href="<spring:theme code=styleSheet/>"
type="text/css"/>
    </head>
    <body style="background=<spring:theme code=background/>">
        ...
    </body>
</html>
```

Por defecto, ResourceBundleThemeSource utiliza un prefijo de nombres base vacío. Como resultado, los archivos de propiedades se cargan desde la raíz de la ruta de clases. Así que hay que poner la definición del tema cool.properties en un directorio de la raíz de la ruta de clases, por ejemplo en /WEB-INF/classes. El ResourceBundleThemeSource utiliza el mecanismo de

carga de recursos Java, lo que permite la plena internacionalización de los temas. Por ejemplo, podríamos tener un /WEB-INF/classes/cool_nl.properties que hace referencia a una imagen especial de fondo con el texto en lengua holandesa en él.

9.3. Resolucion de temática

El DispatcherServlet buscará un bean llamado themeResolver para saber qué implementación ThemeResolver va a utilizar. Una resolución temática funciona de la misma manera que un LocaleResolver, detecta el tema a usar para una petición concreta y también puede cambiar el tema de la solicitud. Los siguientes resolvers temáticos son proporcionados Spring:

Clase	Descripción
FixedThemeResolver	Selecciona un tema fijo, establecido mediante la propiedad default ThemeName.
FixedThemeResolver	El tema se mantiene en la sesión HTTP del usuario. Sólo debe establecerse una vez por cada sesión, pero no se conserva entre las sesiones.
CookieThemeResolver	El tema seleccionado se almacena en una cookie en el cliente.

Figura 27. Resolvers temáticos de Spring.

Spring también proporciona una ThemeChangeInterceptor que permite cambios temáticos en cada petición con un parámetro de solicitud simple.

10. Spring multipart support

10.1. Introducción

Spring cuenta con un soporte integrado de carga de archivos divididos en varias partes para aplicaciones web. Para habilitar este soporte se utilizan objetos MultipartResolver, definidos en el paquete org.springframework.web.multipart. Spring proporciona una implementación MultipartResolver para ser usada junto con Commons FileUpload y otro para usarse con una solicitud Servlet 3.0 multipartes.

Por defecto Spring no utiliza archivos multipartes, debido a que algunos desarrolladores quieren manejarlos por sí mismos. Para habilitar este manejo es necesario añadir una resolución de multipartes en el contexto de la aplicación web. Cada solicitud es inspeccionada para ver si contiene varias partes. Si no se encuentra ninguna, la solicitud continúa como se esperaba. Si se encuentran varias partes en la solicitud, el MultipartResolver que se ha declarado en el contexto se utiliza. Después de eso, el atributo de multipartes en su solicitud se trata como cualquier otro atributo.

10.2. Usar un MultipartResolver con *Commons FileUpload*

El siguiente ejemplo muestra cómo utilizar el CommonsMultipartResolver:

```
<bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResol
ver">

    <!-- one of the properties available; the maximum file size in
bytes -->
    <property name="maxUploadSize" value="100000"/>

</bean>
```

Por supuesto, también hay que poner los .jar apropiados en la ruta de clases para trabajar con la resolución multipartes. En el caso de CommonsMultipartResolver, es necesario utilizar commons-fileupload.jar.

10.3. Usar un MultipartResolver con *Servlet 3.0*

Para utilizar Servlet 3.0 con multipartes, es necesario marcar el DispatcherServlet con un "multipart-config" en la sección correspondiente de web.xml, o con un javax.servlet.MultipartConfigElement en el registro de Servlets, o en caso de una clase Servlet personalizada con una anotación javax.servlet.annotation.MultipartConfig en su clase Servlet. Tanto las opciones de configuración como los tamaños máximos, o lugares de almacenamiento deben ser indicados a nivel de registro de Servlet.

Una vez que Servlet 3.0 de multipartes se ha habilitado de una de las formas antes mencionadas se puede agregar el StandardServletMultipartResolver a la configuración Spring:

```
<bean id="multipartResolver"

class="org.springframework.web.multipart.support.StandardServletMultip
artResolver">
</bean>
```

10.4. Carga de un archivo en un formulario

Después de la MultipartResolver complete su trabajo, la solicitud es procesada como cualquier otra. En primer lugar, crear un formulario con un archivo de entrada que le permitirá al usuario cargar un formulario. El atributo de codificación (enctype ="multipart/form-data") permite que el navegador sepa cómo codificar el formulario como una petición de varias partes:

```
<html>
    <head>
        <title>Upload a file please</title>
    </head>
```

```
    <body>
        <h1>Please upload a file</h1>
        <form method="post" action="/form" enctype="multipart/form-
data">
            <input type="text" name="name"/>
            <input type="file" name="file"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

El siguiente paso es crear un controlador que se encargue de la carga de archivos. Este controlador es muy similar a uno normal anotado con @Controller, excepto que usamos MultipartHttpServletRequest o MultipartFile en los parámetros del método:

```
@Controller
public class FileUploadController {

    @RequestMapping(value = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name,
            @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }

        return "redirect:uploadFailure";
    }}
```

Cuando se utiliza Servlet 3.0 con multipartes también se puede utilizar javax.servlet.http.Part como parámetro del método:

```
@Controller
public class FileUploadController {

    @RequestMapping(value = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name,
            @RequestParam("file") Part file) {

        InputStream inputStream = file.getInputStream();
        // store bytes from uploaded file somewhere

        return "redirect:uploadSuccess";
    }
}
```

11. Manejo de excepciones

11.1. HandlerExceptionResolver

Las implementaciones HandlerExceptionResolver que proporciona Spring MVC tratan excepciones inesperadas que se producen durante la ejecución del controlador. HandlerExceptionResolver se parece bastante al mapeo de excepción que se pueden definir en el descriptor de la aplicación web web.xml. Sin embargo, proporcionan una forma más flexible de hacerlo. Por ejemplo proporcionando información sobre qué controlador se estaba ejecutando cuando se produjo la excepción. Por otra parte, una manera de manejar las excepciones da más opciones para responder adecuadamente ante la solicitud que se reenvía a otra dirección URL.

Además de la implementación de la interfaz HandlerExceptionResolver, que es sólo una parte de la implementación del método resolveException (Exception, Handler) y devuelve un ModelAndView, también se puede utilizar SimpleMappingExceptionResolver o crear métodos @ExceptionHandler. SimpleMappingExceptionResolver permite tomar el nombre de cualquier clase de excepción que se pueda arrojar y asignarla a un nombre de vista. Esto es funcionalmente equivalente a la función de mapeo de excepción de la API de Servlet, pero también es posible implementar asignaciones más finamente granuladas de excepciones de diferentes controladores. Por otro lado, la anotación @ExceptionHandler se puede utilizar en los métodos que deben invocarse para controlar una excepción, tales métodos, pueden ser definidos a nivel local dentro de un @Controller o pueden aplicarse a muchas clases @Controller definiéndose dentro de una clase @ControllerAdvice.

11.2. @ExceptionHandler

La interfaz HandlerExceptionResolver y las implementaciones SimpleMappingExceptionResolver permiten asignar excepciones a vistas específicas. Sin embargo, en algunos casos, especialmente cuando se basan en métodos @responseBody en lugar de resolución de vistas, puede ser más conveniente establecer directamente el estado de la respuesta y, opcionalmente, escribir el contenido de error para el cuerpo de la respuesta.

Esto puede hacerse con métodos @ExceptionHandler. Cuando @ExceptionHandler se declara en un controlador los tipos de métodos se aplican en las excepciones planteadas por métodos @RequestMapping de ese controlador. También se puede declarar un método @ExceptionHandler dentro de una clase @ControllerAdvice, en cuyo caso manejará excepciones de los métodos @RequestMapping de muchos controladores. A continuación se muestra un ejemplo de un método @ExceptionHandler:

```
@Controller
public class SimpleController {

    // @RequestMapping methods omitted ...

    @ExceptionHandler(IOException.class)
    public ResponseEntity<String> handleIOException(IOException ex) {
```

```
        // prepare responseEntity
        return responseEntity;
    }

}
```

El valor @ExceptionHandler puede ajustarse a una gran variedad de tipos de excepción. Si se produce una excepción que coincide con uno de los tipos en la lista, entonces el método anotado con @ExceptionHandler será invocado. Si el valor de la anotación no se establece, se utilizaran los tipos de excepción que figuran como argumentos del método.

Al igual que los métodos de controlador estándar con una anotación @RequestMapping, los argumentos de los métodos y los valores de retorno de los métodos @ExceptionHandler pueden ser flexibles.

11.3. Controlador de excepciones estándar

Spring MVC puede plantear una serie de excepciones al procesar una solicitud. El SimpleMappingExceptionResolver puede asignar fácilmente cualquier excepción a una vista de error por defecto según sea necesario. Dependiendo de la excepción que se dio el código de estado puede indicar un error de cliente (4xx) o un error de servidor (5xx).

El DefaultHandlerExceptionResolver traduce excepciones Spring MVC a códigos de estado de error específicos. Se registra por defecto con el espacio de nombres de MVC, la config MVC Java, y el DispatcherServlet . A continuación se mencionan algunas de las excepciones manejadas por esta resolución y los códigos de estado correspondientes:

Excepción	**Código de estado HTTP**
BindException	400 (Bad Request)
ConversionNotSupportedException	500 (Internal Server Error)
HttpMediaTypeNotAcceptableException	406 (no aceptable)
HttpMediaTypeNotSupportedException	415 (Unsupported Media Type)
HttpMessageNotReadableException	400 (Bad Request)
HttpMessageNotWritableException	500 (Internal Server Error)
HttpRequestMethodNotSupportedException	405 (método no permitido)
MethodArgumentNotValidException	400 (Bad Request)

Excepción	Código de estado HTTP
MissingServletRequestParameterException	400 (Bad Request)
MissingServletRequestPartException	400 (Bad Request)
NoHandlerFoundException	404 (Not Found)
NoSuchRequestHandlingMethodException	404 (Not Found)
TypeMismatchException	400 (Bad Request)

Figura 28. Tipos de excepciones.

DefaultHandlerExceptionResolver funciona de forma transparente mediante el establecimiento de la situación de la respuesta. Sin embargo, no llega a escribir ningún mensaje del error en el cuerpo de la respuesta, mientras que la aplicación puede tener que agregar contenido para cada respuesta de error, por ejemplo, cuando se proporciona una API REST.

11.4. Anotar excepciones con @ResponseStatus

Una excepción puede ser anotada con @ResponseStatus. Cuando se produce la excepción, ResponseStatusExceptionResolver la maneja estableciendo el estado de la respuesta en consecuencia. Por defecto, el DispatcherServlet registra el ResponseStatusExceptionResolver, por lo que está disponible para su uso.

11.5. Personalizar por defecto una página de error

Cuando el estado de la respuesta se establece en un código de estado de error y el cuerpo de la respuesta está vacío, los contenedores de Servlets comúnmente redireccionan a una página de error con formato HTML. Para personalizar la página de error por defecto del contenedor, se puede declarar un elemento <error-page> en web.xml. En Servlet 3 una página de error no tiene por qué ser mapeada, lo que significa que la ubicación especificada personaliza la página de error por defecto.

```
<error-page>
    <location>/error</location>
</error-page>
```

Al escribir información de error, se puede acceder al código de estado y al mensaje de error establecido en el HttpServletResponse a través de los atributos de la petición en un controlador:

```
@Controller
public class ErrorController {

    @RequestMapping(value="/error", produces="application/json")
    @ResponseBody
    public Map<String, Object> handle(HttpServletRequest request) {

        Map<String, Object> map = new HashMap<String, Object>();
        map.put("status",
request.getAttribute("javax.servlet.error.status_code"));
        map.put("reason",
request.getAttribute("javax.servlet.error.message"));

        return map;
    }

}
```

O en un JSP:

```
<%@ page contentType="application/json" pageEncoding="UTF-8"%>
{
    status:<%=request.getAttribute("javax.servlet.error.status_code")
%>,
    reason:<%=request.getAttribute("javax.servlet.error.message") %>
}
```

12. Convención sobre la configuración

Para una gran cantidad de proyectos, apegarse a las convenciones establecidas y que tengan valores razonables por defecto es justo lo que necesitan, y Spring Web MVC cuenta con un soporte explícito de convención sobre la configuración. Esto significa que si se establece un conjunto de convenciones de nombres y cosas por el estilo, se puede reducir sustancialmente la cantidad de configuración que se requiere para establecer el mapeo del controlador, ver los resolvers, las instancias ModelAndView, etc.

El soporte de convención sobre la configuración aborda las tres áreas principales del MVC: modelos, vistas y controladores.

12.1. El Controlador ControllerClassNameHandlerMapping

La clase ControllerClassNameHandlerMapping es una implementación HandlerMapping que utiliza una convención para determinar la asignación entre direcciones URL de solicitud y los Controlador que van a manejar esas peticiones.

Consideremos el siguiente Controlador de una aplicación.

```
public class ViewShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) {
        // the implementation is not hugely important for this
example...
    }

}
```

Aquí se muestra un fragmento del archivo de configuración de Spring Web MVC:

```
<bean
class="org.springframework.web.servlet.mvc.support.ControllerClassName
HandlerMapping"/>

<bean id="viewShoppingCart" class="x.y.z.ViewShoppingCartController">
    <!-- inject dependencies as required... -->
</bean>
```

ControllerClassNameHandlerMapping encuentra todos los controladores de beans definidos en su contexto de aplicación y define la asignación del controlador. Por lo tanto, ViewShoppingCartController se asigna a la solicitud de URL/viewshoppingcart*.

A continuación veremos algunos ejemplos:

- WelcomeController se asigna a la solicitud de URL /welcome*.
- HomeController se asigna a la solicitud de URL /home*.
- IndexController se asigna a la solicitud de URL /index*.

12.2. El Modelo ModelMap (ModelAndView)

La clase ModelMap es esencialmente un Map que puede hacer que la adición de los objetos que se van a mostrar en una vista se adhieran a una convención. Vamos a considerar el siguiente controlado.

```
public class DisplayShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) {
```

```
        List cartItems = // get a List of CartItem objects
        User user = // get the User doing the shopping

        ModelAndView mav = new ModelAndView("displayShoppingCart"); <-
- the logical view name

        mav.addObject(cartItems); <-- look ma, no name, just the
object
        mav.addObject(user); <-- and again ma!

        return mav;
    }
}
```

La clase ModelAndView utiliza una clase ModelMap, que es una implementación personalizada de Map, que genera automáticamente una clave para un objeto cuando se agrega a la misma. La estrategia para determinar el nombre de un objeto adicional es utilizar el nombre de la clase por debajo de la clase del objeto. Los siguientes ejemplos son nombres que se generan para objetos puestos en un ejemplo de ModelMap.

- Una instancia xyUser tendrá el nombre de usuario generado.
- Una instancia xyRegistration tendrá el nombre de registro generado.
- Una instancia java.util.HashMap tendrá el nombre del HashMap generado.

12.3. RequestToViewNameTranslator

La interfaz RequestToViewNameTranslator determina un nombre de vista cuando se proporciona explícitamente tal nombre de vista. Tiene una sola implementación, la clase DefaultRequestToViewNameTranslator.

DefaultRequestToViewNameTranslator solicita direcciones URL a los nombres de las vistas, como en el siguiente ejemplo:

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) {
        // process the request...
        ModelAndView mav = new ModelAndView();
        // add data as necessary to the model...
        return mav;
        // notice that no View or logical view name has been set
    }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- this bean with the well known name generates view names for
us -->
    <bean id="viewNameTranslator"

class="org.springframework.web.servlet.view.DefaultRequestToViewNameTr
anslator"/>

    <bean class="x.y.RegistrationController">
        <!-- inject dependencies as necessary -->
    </bean>

    <!-- maps request URLs to Controller names -->
    <bean
class="org.springframework.web.servlet.mvc.support.ControllerClassName
HandlerMapping"/>

    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolv
er">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

En la implementación del método handleRequest (..) ni la vista ni el nombre de la vista han sido asignadas en el ModelAndView devuelto. El DefaultRequestToViewNameTranslator tiene la tarea de generar un nombre de vista para la dirección URL de la solicitud. En el caso anterior, el RegistrationController se utiliza en conjunción con el ControllerClassNameHandlerMapping, una solicitud de URL de los resultados de http://localhost/registration.html y un nombre de vista del registro que se generan mediante el DefaultRequestToViewNameTranslator. Este nombre de vista es resuelto en la vista /WEB-INF/jsp/registration.jsp por el bean InternalResourceViewResolver.

13. Configuración de Spring MVC

En esta sección se introducirán dos maneras de configurar Spring MVC: la configuración MVC de Java y el espacio de nombres MVC con XML.

La configuración MVC de Java y el espacio de nombres MVC proporcionan una configuración predeterminada similar a la de los DispatcherServlet y que anula sus valores predeterminados. El objetivo es evitar que la mayoría de las aplicaciones tengan que crear la misma configuración y proporcionar un punto de partida simple y que requiere poco o ningún conocimiento previo de la configuración subyacente mediante constructores de alto nivel para la configuración de Spring MVC.

Se puede elegir la configuración MVC de Java o el espacio de nombres MVC dependiendo de la preferencia del desarrollador. Además, como se verá más adelante, con la configuración

MVC de Java es más fácil ver la configuración subyacente, así como para realizar personalizaciones de grano fino directamente sobre los beans de Spring MVC.

13.1. Habilitar la configuración MVC de Java o el espacio de nombres MVC con XML

Para activar configuración MVC de Java hay que añadir la anotación @EnableWebMvc a una de las clases @Configuration:

```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

Para lograr lo mismo con XML hay que utilizar el elemento mvc:annotation-driven:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven />
</beans>
```

13.2. Personalización de la configuración

Para personalizar la configuración por defecto de Java sólo se tiene usar la interfaz WebMvcConfigurer o extender la clase WebMvcConfigurerAdapter y reemplazar los métodos que se necesiten. A continuación se muestra un ejemplo de algunos de los métodos disponibles.

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    protected void addFormatters(FormatterRegistry registry) {
        // Add formatters and/or converters
    }

    @Override
    public void
configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        // Configure the list of HttpMessageConverters to use
    }
```

```
}
```

Para personalizar la configuración por defecto de <mvc:annotation-driven> hay que comprobar los atributos y los subelementos soportados. Se puede utilizar el esquema XML Spring MVC o la función de completado de código del IDE para descubrir que atributos y subelementos están disponibles. El siguiente ejemplo muestra un subconjunto que está disponible:

```
<mvc:annotation-driven conversion-service="conversionService">
    <mvc:message-converters>
        <bean class="org.example.MyHttpMessageConverter"/>
        <bean class="org.example.MyOtherHttpMessageConverter"/>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="conversionService"
class="org.springframework.format.support.FormattingConversionServiceF
actoryBean">
    <property name="formatters">
        <list>
            <bean class="org.example.MyFormatter"/>
            <bean class="org.example.MyOtherFormatter"/>
        </list>
    </property>
</bean>
```

13.3. Configuración de interceptores

Se pueden configurar HandlerInterceptors o WebRequestInterceptors para que se apliquen a todas las solicitudes entrantes o restringirlos a determinados patrones de URL.

Un ejemplo de interceptores registrados en Java:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleInterceptor());
        registry.addInterceptor(new
ThemeInterceptor()).addPathPatterns("/").excludePathPatterns("/admin/"
);
        registry.addInterceptor(new
SecurityInterceptor()).addPathPatterns("/secure/*");
    }

}
```

En XML hay que utilizar el elemento <mvc:interceptors>:

```
<mvc:interceptors>
    <bean
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
/>
    <mvc:interceptor>
        <mvc:mapping path="/"/>
        <mvc:exclude-mapping path="/admin/"/>
        <bean
class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"
/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/secure/*"/>
        <bean class="org.example.SecurityInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>
```

13.4. Configuración del contenido

También se puede configurar el modo en el que Spring MVC determina los tipos solicitados por parte del cliente para el mapeo de solicitud, así como para fines de negociación de contenido. Las opciones disponibles para comprobar la extensión del archivo en el URI de la solicitud son la cabecera "Accept" y un parámetro de la petición. Por defecto, la extensión de archivo en la petición URI se comprueba primero y la cabecera "Accept" se comprueba a continuación.

Para las extensiones de archivo de la solicitud de URI, la config MVC Java y el espacio de nombres MVC, registran automáticamente extensiones como .json, .xml, .rss y .atom, si las dependencias correspondientes; como Jackson, jaxb2 o Rome; están presentes en el classpath. Extensiones adicionales pueden no necesitar registrarse de manera explícita si se pueden descubrir a través de ServletContext.getMimeType(String) o el Activation Framework Java. Se pueden registrar más extensiones con el método setUseRegisteredSuffixPatternMatch.

A continuación se muestra un ejemplo de la personalización del contenido a través de la configuración MVC Java:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void
configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.favorPathExtension(false).favorParameter(true);
    }
}
```

En el espacio de nombres MVC, el elemento <mvc:annotation-driven> tiene un atributo content-negotiation-manager, que espera un ContentNegotiationManager que se puede crear con un ContentNegotiationManagerFactoryBean :

```
<mvc:annotation-driven content-negotiation-
manager="contentNegotiationManager" />

<bean id="contentNegotiationManager"
class="org.springframework.web.accept.ContentNegotiationManagerFactory
Bean">
    <property name="favorPathExtension" value="false" />
    <property name="favorParameter" value="true" />
    <property name="mediaTypes" >
        <value>
            json=application/json
            xml=application/xml
        </value>
    </property>
</bean>
```

Si no se utiliza la configuración MVC de Java o el espacio de nombres MVC, se tendrá que crear una instancia de ContentNegotiationManager y utilizarla para configurar RequestMappingHandlerMapping para resolver solicitudes y RequestMappingHandlerAdapter y ExceptionHandlerExceptionResolver para fines de contenido.

13.5. Configuración de los controladores de la vista

Este es un atajo para definir un ParameterizableViewController que inmediatamente remite a una vista cuando se invoca. Es recomendable usarlo en casos estáticos, cuando no hay lógica del controlador Java para ejecutarse antes de la vista que genera la respuesta.

Un ejemplo de la transmisión de una solicitud "/" a una vista llamada "home" en Java:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```

Y lo mismo en XML utilizando el elemento <mvc:view-controller>:

```
<mvc:view-controller path="/" view-name="home"/>
```

13.6. Configuración de recursos

Esta opción permite que las solicitudes de recursos estáticos sigan un patrón de URL en particular para ser utilizados por un ResourceHttpRequestHandler de cualquier lista de recursos. Esto proporciona una manera conveniente para proporcionar recursos estáticos desde ubicaciones distintas a la raíz de la aplicación web, incluyendo sitios de la ruta de clases. La propiedad cache-period se puede utilizar para establecer la espiración de los encabezados para que puedan ser utilizados de manera más eficiente por parte del cliente. El controlador también evalúa adecuadamente la cabecera Last-Modified para que un código 304 de estado sea devuelto cuando sea necesario, y evitar una sobrecarga innecesaria de los recursos que se almacenan en caché por el cliente. Por ejemplo, para atender las solicitudes de recursos con un patrón de URL /resources/ ** a partir de los recursos públicos de un directorio dentro de la raíz que la aplicación web debería utilizar:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
{
registry.addResourceHandler("/resources/**").addResourceLocations("/pu
blic-resources/");
}}
```

Y lo mismo en XML:

```
<mvc:resources mapping="/resources/**" location="/public-resources/"/>
```

Para permitir que estos recursos expiren en un periodo de 1 año, para asegurar el uso máximo de la caché del navegador y una reducción en las solicitudes HTTP realizadas por el navegador, se utiliza el siguiente fragmento:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
{

registry.addResourceHandler("/resources/**").addResourceLocations("/pu
blic-resources/").setCachePeriod(31556926);
    }
}
```

Y en XML:

```
<mvc:resources mapping="/resources/**" location="/public-resources/"
cache-period="31556926"/>
```

El atributo mapping debe ser un patrón de Ant que puede ser utilizado por un SimpleUrlHandlerMapping, y la ubicación del atributo se deben especificar en uno o más directorios de recursos válidos. Las ubicaciones de recursos múltiples pueden especificarse usando una lista separando los valores por comas. Los puntos que se indican serán revisados en el orden especificado por el recurso para cualquier solicitud. Por ejemplo, para permitir que los recursos tanto de la raíz de la aplicación web y de una dirección conocida como /META-INF/public-web-resources/ en cualquier .jar en la ruta de clase:

```
@EnableWebMvc
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
{
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/", "classpath:/META-
INF/public-web-resources/");
    }
}
```

Y en XML:

```
<mvc:resources mapping="/resources/**" location="/, classpath:/META-
INF/public-web-resources/"/>
```

A la hora de proporcionar recursos que pueden cambiar cuando se implementa una nueva versión de la aplicación, se recomienda que se incorpore un String en el patrón de mapeo utilizado para solicitar los recursos, de modo que se pueda obligar a los clientes a solicitar la versión más reciente de su solicitud de recursos.

Como ejemplo, consideremos una aplicación que utiliza en la producción una versión personalizada de rendimiento optimizado de la biblioteca de Dojo JavaScript, y que en la construcción se desplegó dentro de la aplicación web en la ruta /public-resources/dojo/dojo.js. Desde diferentes partes de Dojo se pueden incorporar nuevos recursos para cada nueva versión de la aplicación, los navegadores web de los clientes deben estar obligados a volver a descargar ese dojo.js. Una forma sencilla de conseguir esto sería gestionar la versión de la aplicación en un archivo de propiedades, tales como:

Application.Version = 1.0.0

Y luego hacer que los valores del archivo de propiedades accesibles mediante Spel sean como un bean mediante la etiqueta util:properties:

```
<util:properties id="applicationProps" location="/WEB-
INF/spring/application.properties"/>
```

Con la versión de la aplicación, ahora accesible a través de Spel, podemos incorporar esto en la etiqueta resources:

```
<mvc:resources mapping="/resources-
#{applicationProps[application.version]}/**" location="/public-
resources/"/>
```

En Java, se puede utilizar la anotación @PropertySouce y luego inyectar la abstracción Environment para acceder a todas las características:

```
@Configuration
@EnableWebMvc
@PropertySource("/WEB-INF/spring/application.properties")
public class WebConfig extends WebMvcConfigurerAdapter {

    @Inject Environment env;

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
{
        registry.addResourceHandler(
                "/resources-" + env.getProperty("application.version")
+ "/**")
                .addResourceLocations("/public-resources/");
    }}
```

Y finalmente, para solicitar el recurso con el URL, podemos tomar ventaja de las etiquetas JSP de Spring:

```
<spring:eval expression="@applicationProps[application.version]"
var="applicationVersion"/>

<spring:url value="/resources-{applicationVersion}" var="resourceUrl">
    <spring:param name="applicationVersion"
value="${applicationVersion}"/>
</spring:url>

<script src="${resourceUrl}/dojo/dojo.js" type="text/javascript">
</script>
```

13.7. MVC: default-servlet-handler

Esta etiqueta permite asignar el DispatcherServlet a "/" (anulando así el mapeo de servlet predeterminado del contenedor), al tiempo que permite que las solicitudes de recursos estáticos sean manejadas por un Servlet predeterminado del contenedor. Se configura un DefaultServletHttpRequestHandler y se asigna una dirección URL "/ **" y la prioridad más baja en relación con otras asignaciones de URL.

Este controlador reenviará todas las peticiones al Servlet por defecto. Por eso es importante que siga siendo el último en el orden de todos los demás URLHandlerMappings.

Para activar la función mediante el uso de la configuración por defecto:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void
configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }

}
```

O en XML:

```
<mvc:default-servlet-handler/>
```

La advertencia de que se sobrescribe la asignación de "/" como Servlet es que el RequestDispatcher para el servlet por defecto debe ser recuperado por su nombre en lugar de por su ruta. El DefaultServletHttpRequestHandler intentará detectar automáticamente el Servlet por defecto para el contenedor al inicio, utilizando una lista de nombres conocidos para los grandes contenedores de Servlets (incluyendo Tomcat, Espolón, GlassFish, JBoss, Resin, WebLogic y WebSphere). El valor por defecto del nombre del Servlet debe ser proporcionada de forma explícita si el Servlet predeterminado se ha configurado de forma personalizada con un nombre diferente, o si un contenedor de Servlets diferente se utiliza cuando no se conoce el nombre del Servlet por defecto, como en el siguiente ejemplo:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void
configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable("myCustomDefaultServlet");
    }
```

```
}
```

O en XML:

```
<mvc:default-servlet-handler default-servlet-
name="myCustomDefaultServlet"/>
```

13.8. Personalización avanzada con MVC Java Config

Como se puede ver en los ejemplos anteriores, config MVC Java y el espacio de nombres MVC proporcionan constructores de alto nivel que no requieren de un conocimiento profundo de los beans subyacentes creados. En su lugar, ayuda a centrarse en las necesidades de la aplicación. Sin embargo, en algún momento es posible que se necesite más control.

El primer paso hacia un mayor control es ver los beans subyacentes creados. En MVC config Java se puede ver el Javadoc y los métodos @beans en WebMvcConfigurationSupport. La configuración de esta clase se importa automáticamente a través de la anotación @EnableWebMvc.

El siguiente paso es personalizar uno de los beans creados en WebMvcConfigurationSupport. Esto requiere dos cosas, quitar la anotación @EnableWebMvc con el fin de impedir la importación y luego extender desde DelegatingWebMvcConfiguration una subclase de WebMvcConfigurationSupport. He aquí un ejemplo:

```
@Configuration
public class WebConfig extends DelegatingWebMvcConfiguration {

    @Override
    public void addInterceptors(InterceptorRegistry registry){
        // ...
    }

    @Override
    @Bean
    public RequestMappingHandlerAdapter requestMappingHandlerAdapter()
{
        // Create or let "super" create the adapter
        // Then customize one of its properties
    }

}
```

4.13.9. Personalización avanzada con el espacio de nombres

Obtener un mayor control sobre una configuración creada es un poco más difícil con el espacio de nombres MVC.

Si no se necesita hacer eso, en lugar de replicar la configuración que se proporciona, hay que configurar un BeanPostProcessor que detecte el bean que se desea personalizar y luego modificar sus propiedades según sea necesario. Por ejemplo:

```
@Component
public class MyPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String
name) throws BeansException {
        if (bean instanceof RequestMappingHandlerAdapter) {
            // Modify properties of the adapter
        }
    }

}
```

Hay que tener en cuenta que MyPostProcessor necesita ser incluido en un <component scan/> con el fin de que sea detectado.

TECNOLOGÍAS DE LAS VISTAS

1. Introducción

Una de las áreas en las que la Spring sobresale es en la separación de las vistas del resto de componentes del framework MVC. Este capítulo trata de las principales tecnologías de vistas que trabajan con Spring.

2. JSP y JSTL

Spring proporciona un par de soluciones out-of-the-box para las vistas JSP y JSTL. El uso de JSP o JSTL se hace mediante una resolución de vistas especificada en el WebApplicationContext.

2.1. Resolución de vistas

Al igual que con cualquier otra tecnología de vistas que esté integrada con Spring, JSP necesita una resolución de vistas que resuelva sus vistas. Los resolutores de vistas más utilizados en el desarrollo de JSP son los InternalResourceViewResolver y los ResourceBundleViewResolver. Ambos se declaran en el WebApplicationContext:

```
<!-- the ResourceBundleViewResolver -->
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver
">
    <property name="basename" value="views"/>
</bean>

# And a sample properties file is uses (views.properties in WEB-
INF/classes):
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

productList.(class)=org.springframework.web.servlet.view.JstlView
productList.url=/WEB-INF/jsp/productlist.jsp
```

Como se puede ver, el ResourceBundleViewResolver necesita un archivo de propiedades que defina los nombres de las vistas asignadas a una clase y a una URL. Con un ResourceBundleViewResolver es posible mezclar diferentes tipos de vistas usando sólo un resolutor.

El InternalResourceBundleViewResolver puede ser configurado para el uso de JSP como se describe a continuación.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolv
er">
    <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
```

```
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

2.2. Biblioteca de etiquetas

Spring ofrece un amplio conjunto de etiquetas de enlace de datos para el manejo de los elementos de un formulario utilizando JSP. Cada etiqueta proporciona soporte para el conjunto de atributos de su correspondiente etiqueta HTML, por lo que el uso de las etiquetas es familiar e intuitivo.

A diferencia de otras bibliotecas de etiquetas, esta se integra con Spring Web MVC, dando a las etiquetas acceso al objeto comando y a los datos de referencia de sus controladores. Como se verá en los siguientes ejemplos, las etiquetas hacen fáciles de desarrollar, leer y mantener a los JSP.

Configuración

La biblioteca de etiquetas incluida en spring-webmvc.jar y el descriptor de la biblioteca se llaman spring-form.tld.

Para utilizar las etiquetas de esta biblioteca, hay que agregar la siguiente directiva en la parte superior de la página JSP:

```
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form" %>
```

Donde “form” es el nombre de prefijo de etiqueta que se desea utilizar para las etiquetas de esta biblioteca.

La etiqueta form

Esta etiqueta representa a una etiqueta HTML y expone una ruta de enlace para las etiquetas internas. Pone el objeto de comando en el PageContext de modo que se puede acceder al objeto mediante etiquetas interiores.

Supongamos que tenemos un objeto de dominio llamado user. Es un JavaBean con propiedades como firstName y lastName. Lo usaremos como objeto para que el controlador de forma devuelva un form.jsp. A continuación se muestra un ejemplo del form.jsp:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
```

```
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

Los valores firstName y lastName se recuperan del objeto comando que se coloca en el PageContext mediante el controlador de la página.

El HTML generado se parece a una form estándar:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text"
value="Harry"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text"
value="Potter"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form>
```

El JSP anterior asume que el nombre de la variable del objeto es 'comand'. Si por el contrario se ha puesto el objeto en el modelo con otro nombre se puede enlazar el formulario a la variable de este modo:

```
<form:form commandName="user">
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
```

```
</form:form>
```

La etiqueta checkbox

Esta etiqueta referencia a una etiqueta HTML de tipo checkbox.

Vamos a suponer que el usuario tiene como preferencias una circular y una lista de aficiones. A continuación se muestra un ejemplo de la clase Preferences:

```
public class Preferences {

    private boolean receiveNewsletter;
    private String[] interests;
    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

El form.jsp resultante sería:

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --
%>
            <td><form:checkbox
path="preferences.receiveNewsletter"/></td>
        </tr>

        <tr>
            <td>Interests:</td>
            <%-- Approach 2: Property is of an array or of type
java.util.Collection --%>
```

```
            <td>
                Quidditch: <form:checkbox path="preferences.interests"
value="Quidditch"/>
                Herbology: <form:checkbox path="preferences.interests"
value="Herbology"/>
                Defence Against the Dark Arts: <form:checkbox
path="preferences.interests" value="Defence Against the Dark Arts"/>
            </td>
        </tr>

        <tr>
            <td>Favourite Word:</td>
            <%-- Approach 3: Property is of type java.lang.Object --%>
            <td>
                Magic: <form:checkbox path="preferences.favouriteWord"
value="Magic"/>
            </td>
        </tr>
    </table>
</form:form>
```

A continuación se muestra un fragmento del código HTML de algunas casillas de verificación:

```
<tr>
    <td>Interests:</td>
    <td>
        Quidditch: <input name="preferences.interests" type="checkbox"
value="Quidditch"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Herbology: <input name="preferences.interests" type="checkbox"
value="Herbology"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Defence Against the Dark Arts: <input
name="preferences.interests" type="checkbox" value="Defence Against
the Dark Arts"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
</tr>
```

La etiqueta checkboxes

Esta etiqueta referencia a múltiples etiquetas HTML de tipo checkbox.

Basándonos en el ejemplo de la etiqueta anterior, a veces uno preferiría no tener que enumerar todas las posibles aficiones en su página JSP. Es preferible proporcionar una lista en tiempo de ejecución de las opciones disponibles y añadirlas a la etiqueta. Ese es el propósito de esta etiqueta. Se pasa en una matriz, una lista o un mapa que contenga las opciones disponibles en la propiedad "items". A continuación se muestra un ejemplo de un JSP utilizando esta etiqueta:

```
<form:form>
    <table>
```

```
        <tr>
            <td>Interests:</td>
            <td>
                <%-- Property is of an array or of type
java.util.Collection --%>
                <form:checkboxes path="preferences.interests"
items="${interestList}"/>
            </td>
        </tr>
    </table>
</form:form>
```

En este ejemplo se supone que el "interestList" es una lista disponible como un atributo del modelo que contiene los valores en forma de cadenas que pueden ser seleccionadas. En el caso de que se utilice un mapa, la clave del mapa se utilizará como el valor y el valor del mapa se utiliza como la etiqueta que se mostrará.

La etiqueta radiobutton

Esta etiqueta referencia a una etiqueta HTML del tipo radio.

Un uso típico implicará varias variables asociadas a la misma propiedad, pero con valores diferentes.

```
<tr>
    <td>Sex:</td>
    <td>
        Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/>
    </td>
</tr>
```

La etiqueta radiobuttons

Esta etiqueta referencia a múltiples etiquetas HTML de tipo de radio.

Al igual que las etiquetas checkboxes anteriormente mencionadas, es posible que desee pasar las opciones disponibles como una variable en tiempo de ejecución. Se referencia una matriz, una lista o un mapa que contiene las opciones disponibles en la propiedad "items". En el caso en que se utilice un mapa, la clave del mapa se utilizará como el valor y el valor del mapa se utiliza como la etiqueta que se mostrará.

```
<tr>
    <td>Sex:</td>
    <td><form:radiobuttons path="sex" items="${sexOptions}"/></td>
</tr>
```

La etiqueta password

Esta etiqueta referencia a una etiqueta HTML del tipo de password.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" />
    </td>
</tr>
```

Por defecto el valor de la contraseña se no se muestra, si se desea que el valor de la contraseña que se muestre hay que establecer el valor del atributo 'showPassword' como true.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq"
showPassword="true" />
    </td>
</tr>
```

La etiqueta select

Esta etiqueta referencia a un elemento HTML select. Es compatible con el enlace de datos de la opción seleccionada, así como el uso anidado de option y de las etiquetas options.

Supongamos un usuario tiene una lista de habilidades.

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
</tr>
```

Si la habilidad del usuario fuese Herbology, la fuente HTML de la fila de habilidades se vería así:

```
<tr>
    <td>Skills:</td>
    <td>
        <select name="skills" multiple="true">
            <option value="Potions">Potions</option>
            <option value="Herbology"
selected="selected">Herbology</option>
            <option value="Quidditch">Quidditch</option>
        </select>
    </td>
</tr>
```

La etiqueta option

Esta etiqueta referencia un elemento HTML option. Establece el atributo select basándose en el valor enlazado.

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

Si el atributo house del usuario es Gryffindor, la fuente HTML de la fila de house se vería así:

```
<tr>
    <td>House:</td>
    <td>
        <select name="house">
            <option value="Gryffindor"
selected="selected">Gryffindor</option>
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
</tr>
```

La etiqueta options

Esta etiqueta referencia una lista de etiquetas HTML option. Establece el atributo select basándose en el valor enlazado.

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code"
itemLabel="name"/>
        </form:select>
    </td>
</tr>
```

Si el usuario vive en el Reino Unido, el código fuente HTML de la fila Country se vería así:

```
<tr>
    <td>Country:</td>
    <td>
        <select name="country">
            <option value="-">--Please Select</option>
            <option value="AT">Austria</option>
            <option value="UK" selected="selected">United
Kingdom</option>
            <option value="US">United States</option>
        </select>
    </td>
</tr>
```

Como muestra el ejemplo, el uso combinado de una etiqueta option con etiqueta options genera el mismo estándar HTML, pero permite especificar explícitamente un valor en el JSP.

La etiqueta textarea

Esta etiqueta referencia un elemento HTML textarea.

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20" /></td>
    <td><form:errors path="notes" /></td>
</tr>
```

La etiqueta error

Esta etiqueta referencia errores de campo en un HTML. Proporciona acceso a los errores creados en el controlador o los que fueron creados por cualquier validador asociado al controlador.

Supongamos que queremos mostrar todos los mensajes de error para los campos firstName y lastName una vez que se envíe el formulario. Tenemos una clase validador de instancias del usuario llamado UserValidator.

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
        return User.class.isAssignableFrom(candidate);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName",
"required", "Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName",
"required", "Field is required.");
    }
}
```

El form.jsp sería:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName" /></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

Si enviamos un formulario con valores vacíos en los campos firstName y lastName, así se vería el código HTML:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is
required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is
required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form>
```

El siguiente ejemplo muestra que la etiqueta error también es compatible con algunas funciones básicas de comodines.

- path = "*" - muestra todos los errores.
- path = " lastName " - muestra todos los errores asociados con el campo lastName.
- si se omite el path - sólo se muestran errores de objeto.

El siguiente ejemplo muestra una lista de errores en la parte superior de la página, seguido de errores de campo específico:

```
<form:form>
    <form:errors path="*" cssClass="errorBox" />
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <td><form:errors path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <td><form:errors path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

El HTML se vería así:

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is
required.<br/>Field is required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <td><span name="firstName.errors">Field is
required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <td><span name="lastName.errors">Field is
required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
</form>
```

3. Tiles

Es posible integrar Tiles en las aplicaciones web que utilizan Spring MVC. A continuación se describe de manera amplia cómo hacer esto.

Para poder utilizar Tiles hay que configurarlo utilizando los archivos que contienen las definiciones. En Spring esto se hace usando el TilesConfigurer. A continuación se muestra un ejemplo de configuración del ApplicationContext:

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>
</bean>
```

Como se puede ver, hay cinco archivos que contienen definiciones, ubicados en el directorio 'WEB-INF/defs'. En la inicialización del WebApplicationContext, los archivos se cargan y se inicializará la fábrica definiciones. Después de que se ha hecho esto, Tiles incluirá los archivos de definición que se pueden utilizar como vistas dentro de la aplicación web. Para poder utilizar las vistas, hay que tener un ViewResolver al igual que con cualquier otra tecnología vista. A continuación se muestran dos posibilidades, la UrlBasedViewResolver y la ResourceBundleViewResolver.

UrlBasedViewResolver

La instancia UrlBasedViewResolver proporciona la clase viewClass para cada vista que tiene que resolver.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass"
value="org.springframework.web.servlet.view.tiles3.TilesView"/>
</bean>
```

ResourceBundleViewResolver

El ResourceBundleViewResolver tiene que ser provisto de un archivo de propiedades que contenga viewnames y viewclasses que el resolver pueda utilizar:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver
">
    <property name="basename" value="views"/>
</bean>
```

```
...
welcomeView.(class)=org.springframework.web.servlet.view.tiles3.TilesV
iew
welcomeView.url=welcome (this is the name of a Tiles definition)

vetsView.(class)=org.springframework.web.servlet.view.tiles3.TilesView
vetsView.url=vetsView (again, this is the name of a Tiles definition)

findOwnersForm.(class)=org.springframework.web.servlet.view.JstlView
findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp
...
```

Como se puede ver, el ResourceBundleViewResolver permitirá combinar diferentes tecnologías de vista.

SimpleSpringPreparerFactory y SpringBeanPreparerFactory

Spring también soporta dos implementaciones especiales de Tiles PreparerFactory.

La especificación SimpleSpringPreparerFactory para una instancia Autowire ViewPreparer se basa en clases específicas de “preparador”, aplicando las llamadas al contenedor así como la configuración del BeanPostProcessors. Si en el contexto se ha activado la configuración de las anotaciones, las anotaciones en las clases ViewPreparer se detectan y se aplican de forma automática.

La especificación SpringBeanPreparerFactory trabaja con un preparador específico de nombres en lugar de uno de clases, obteniendo el bean correspondiente del contexto de aplicación de la DispatcherServlet. El contexto de la aplicación tendrá el control del proceso de creación del bean en este caso.

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>

    <!-- resolving preparer names as Spring bean definition names -->
    <property name="preparerFactoryClass"

value="org.springframework.web.servlet.view.tiles3.SpringBeanPreparerF
actory"/>

</bean>
```

4. Velocity y FreeMarker

Velocity y FreeMarker son dos tipos de plantillas que se pueden utilizar como tecnologías de vistas dentro de las aplicaciones Spring MVC. Las dos son muy similares y tienen necesidades similares, por lo que se tratarán juntas en esta sección.

4.1. Las dependencias

La aplicación web deberá incluir velocity-1.x.x.jar y commons-collections.jar con el fin de trabajar con Velocity o freemarker-2.x.jar para trabajar con FreeMarker. Normalmente se incluyen en la carpeta WEB-INF/lib donde se garantiza que serán encontradas por un servidor Java EE y añadidas a la ruta de clase. Si se usa Spring dateToolAttribute o numberToolAttribute en las vistas Velocity, también habrá que incluir velocity-tools-generic-1.x.jar.

4.2. Configuración del contexto

Para realizar la configuración adecuada es necesario añadir la definición de la configuración del bean en el '*-servlet.xml' como se muestra a continuación:

```
<!--
This bean sets up the Velocity environment for us based on a root path
for templates.
Optionally, a properties file can be specified for more control over
the Velocity
environment, but the defaults are pretty sane for file based template
loading.
-->
<bean id="velocityConfig"
class="org.springframework.web.servlet.view.velocity.VelocityConfigure
r">
    <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
</bean>

<!--
View resolvers can also be configured with ResourceBundles or XML
files. If you need
different view resolving based on Locale, you have to use the resource
bundle resolver.
-->
<bean id="viewResolver"
class="org.springframework.web.servlet.view.velocity.VelocityViewResol
ver">
    <property name="cache" value="true"/>
    <property name="prefix" value=""/>
    <property name="suffix" value=".vm"/>
</bean>
```

```
<!-- freemarker config -->
```

```
<bean id="freemarkerConfig"
class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfi
gurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>

<!--
View resolvers can also be configured with ResourceBundles or XML
files. If you need
different view resolving based on Locale, you have to use the resource
bundle resolver.
-->
<bean id="viewResolver"
class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewR
esolver">
    <property name="cache" value="true"/>
    <property name="prefix" value=""/>
    <property name="suffix" value=".ftl"/>
</bean>
```

4.3. Configuración avanzada

Las configuraciones básicas serán adecuadas para la mayoría de los requisitos de la aplicación, sin embargo las opciones de configuración avanzadas están disponibles para necesidades inusuales.

velocity.properties

Este archivo es completamente opcional, pero si se especifica, contiene los valores que se pasan en tiempo de ejecución a Velocity con el fin de que se configure él mismo. Sólo se requiere para configuraciones avanzadas, pero si se necesita este archivo, hay que especificar su ubicación en la definición del bean VelocityConfigurer.

```
<bean id="velocityConfig"
class="org.springframework.web.servlet.view.velocity.VelocityConfigure
r">
    <property name="configLocation" value="/WEB-
INF/velocity.properties"/>
</bean>
```

Como alternativa, se pueden especificar las propiedades de Velocity directamente en la definición del bean reemplazando la propiedad "configLocation" con las siguientes propiedades en línea.

```
<bean id="velocityConfig"
class="org.springframework.web.servlet.view.velocity.VelocityConfigure
r">
    <property name="velocityProperties">
        <props>
            <prop key="resource.loader">file</prop>
```

```
            <prop key="file.resource.loader.class">

org.apache.velocity.runtime.resource.loader.FileResourceLoader
            </prop>
            <prop key="file.resource.loader.path">${webapp.root}/WEB-
INF/velocity</prop>
            <prop key="file.resource.loader.cache">false</prop>
        </props>
    </property>
</bean>
```

FreeMarker

FreeMarker Settings y SharedVariables pueden ser usadas directamente por el objeto FreeMarker Configuration estableciendo las correspondientes propiedades del bean en el FreeMarkerConfigurer bean. La propiedad freemarkerSettings requiere un objeto java.util.Properties y la propiedad freemarkerVariables requiere un java.util.Map.

```
<bean id="freemarkerConfig"
class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfi
gurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
    <property name="freemarkerVariables">
        <map>
            <entry key="xml_escape" value-ref="fmXmlEscape"/>
        </map>
    </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

4.5. Soporte Bind

Spring proporciona una biblioteca de etiquetas para su uso en JSP que contiene una etiqueta <spring:bind/>. Esta etiqueta permite mostrar los valores de los objetos del formulario y mostrar los resultados de las validaciones fallidas de un validador web de varias formas. Spring tiene un soporte para la misma funcionalidad, tanto para Velocity como para FreeMarker, con macros adicionales para la generación de elementos de entrada del formulario propios.

Los macros bind

Un conjunto estándar de macros se mantienen dentro del archivo spring-webmvc.jar para ambas plantillas, por lo que siempre están disponibles para una aplicación que este configurada como tal.

Algunas de las macros definidas en las bibliotecas de Spring se consideran privadas. Las siguientes secciones se concentraran únicamente en las macros que pueden ser llamadas directamente desde dentro de las plantillas.

Bind simple

En los formularios HTML que actúan como FormView para un controlador de Spring, se puede utilizar código similar al siguiente para enlazar los valores de campo y mostrar mensajes de error para cada campo de entrada de forma similar a un JSP. A continuación se muestra un código de ejemplo para las vistas personFormV y personFormF configuradas anteriormente;

```
<!-- velocity macros are automatically available -->
<html>
    ...
    <form action="" method="POST">
        Name:
        #springBind( "command.name" )
        <input type="text"
            name="${status.expression}"
            value="$!status.value" /><br>
        #foreach($error in $status.errorMessages) <b>$error</b> <br>
#end
        <br>
        ...
        <input type="submit" value="submit"/>
    </form>
    ...
</html>
```

```
<!-- freemarker macros have to be imported into a namespace. We
strongly
recommend sticking to spring -->
<#import "/spring.ftl" as spring />
<html>
    ...
    <form action="" method="POST">
        Name:
        <@spring.bind "command.name" />
        <input type="text"
            name="${spring.status.expression}"
            value="${spring.status.value?default("")}" /><br>
        <#list spring.status.errorMessages as error> <b>${error}</b>
<br> </#list>
        <br>
        ...
        <input type="submit" value="submit"/>
    </form>
    ...
</html>
```

#SpringBind/<@spring.bind> requiere una ruta de argumentos que consiste en el nombre de su objeto de comando seguido de un punto y el nombre del campo que el objeto comando desea enlazar. También se pueden utilizar campos anidados como "command.address.street".

5. XSLT

XSLT es un lenguaje de transformación para XML muy popular dentro de las aplicaciones web. XSLT puede ser una buena opción como una tecnología de vistas si la aplicación trabaja en conjunto con XML, o si el modelo puede ser fácilmente convertido a XML. La siguiente sección muestra cómo producir un documento XML como un modelo de datos y como transformarlo con XSLT en una aplicación Spring Web MVC.

Este ejemplo es una aplicación trivial de Spring que crea una lista de palabras en el controlador y las agrega al mapa de modelo. El mapa se devuelve junto con el nombre de la vista de nuestra vista XSLT.

Definición de un bean

La configuración para una aplicación sencilla es estándar, el fichero de configuración del Dispatcher Servlet contendrá una referencia a un ViewResolver, a asignaciones de URL y a un solo bean del controlador.

```
<bean id="homeController"class="xslt.HomeController"/>
```

Código estándar del controlador MVC

La lógica del controlador se encapsula en una subclase de AbstractControlle, con el método de control que se define:

```
protected ModelAndView handleRequestInternal(HttpServletRequest
request,
        HttpServletResponse response) throws Exception {

    Map map = new HashMap();
    List wordList = new ArrayList();

    wordList.add("hello");
    wordList.add("world");

    map.put("wordList", wordList);

    return new ModelAndView("home", map);
}
```

El modelo de datos se crea de la misma forma que lo haría para cualquier otra aplicación Spring MVC. Dependiendo de la configuración de la aplicación, la lista de palabras del ejemplo podría traducirse por JSP/JSTL haciendo que se añadan como atributos de la solicitud, o que podían ser manejadas por Velocity al añadir el objeto al VelocityContext. Con el fin de tener XSLT, tiene que convertirse en un documento XML de alguna manera. Spring proporciona flexibilidad para crear el DOM de su modelo en cualquier forma.

Convertir los datos del modelo a XML

Con el fin de crear un documento DOM desde una lista de palabras o desde cualquier otro modelo de datos, debemos crear una subclase de la clase org.springframework.web.servlet.view.xslt.AbstractXsltView. Al hacerlo, se debeponer en práctica el método abstracto createXsltSource(..). El primer parámetro pasado a este método es nuestro mapa de modelo. Aquí está la lista completa de clases de la Página de Inicio de nuestra aplicación:

```
package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

    protected Source createXsltSource(Map model, String rootName,
            HttpServletRequest request, HttpServletResponse response)
throws Exception {

        Document document =
DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument(
);
        Element root = document.createElement(rootName);

        List words = (List) model.get("wordList");
        for (Iterator it = words.iterator(); it.hasNext();) {
            String nextWord = (String) it.next();
            Element wordNode = document.createElement("word");
            Text textNode = document.createTextNode(nextWord);
            wordNode.appendChild(textNode);
            root.appendChild(wordNode);
        }
        return new DOMSource(root);
    }

}
```

Una serie de parámetros nombre/valor se pueden definir por la subclase que se agrega al objeto de transformación. Los nombres de los parámetros deben coincidir con los definidos en la plantilla XSLT declarada con <xsl:param name="myParam"> defaultValue </xsl: param> .

Definición de las propiedades de la vista

El archivo views.properties tendrá este aspecto:

home.class)=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

Aquí, se puede ver cómo la vista está vinculada con la clase HomePage que maneja el modelo DOM en la primera propiedad '.(class)'. La propiedad'stylesheetLocation' apunta al archivo XSLT que se encargará de la transformación de XML a HTML para nosotros y la última

propiedad '. root' es el nombre que se usará como la raíz del documento XML. Esto se pasa a la clase HomePage en el segundo parámetro del método createXsltSource(..).

Transformación de documentos

Por último, tenemos el código XSLT utilizado para transformar el documento anterior. Como se muestra en el anterior archivo 'views.properties', el stylesheet se llama 'home.xslt' y se encuentra en el directorio 'WEB-INF/xsl'.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>
                <h1>My First Words</h1>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="word">
        <xsl:value-of select="."/><br/>
    </xsl:template>

</xsl:stylesheet>
```

6. Vistas de documentos (PDF / Excel)

6.1. Introducción

Una página HTML no es siempre la mejor manera para que el usuario vea los resultados del modelo, y la primSpring hace que sea sencillo generar un documento PDF o una hoja de cálculo Excel de forma dinámica a partir de los datos del modelo. El documento es la vista y se transmite desde el servidor con el tipo de contenido correcto hasta el cliente para ejecutar su hoja de cálculo o aplicación de visualización de PDF.

Con el fin de utilizar las vistas Excel, es necesario agregar la biblioteca poi a la ruta de clases, y para la generación de PDF, la biblioteca iText.

6.2. Configuración y setup

Las vistas basadas en documentos se manejan de una manera similar a un a vista XSLT, las secciones siguientes se basan en esto, mediante la demostración de cómo se invoca el mismo

controlador utilizado en el ejemplo XSLT para representar el mismo modelo en un PDF y un Excel.

Ver documentación definiciones

En primer lugar, vamos a modificar el archivo views.properties y añadir la definición de una vista para ambos tipos de documentos. El archivo completo se verá así:

```
home.(class)=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.(class)=excel.HomePage

pdf.(class)=pdf.HomePage
```

Código del controlador

El código del controlador es exactamente el mismo del ejemplo anterior XSLT.

Crear subclases para las vistas Excel

Exactamente como en el ejemplo XSLT, se utilizan las clases abstractas adecuadas con el fin de implementar un comportamiento personalizado en la generación de nuestros documentos de salida. Para Excel, se trata de crear una subclase de org.springframework.web.servlet.view.document.AbstractExcelView (para archivos de Excel generados por POI) o org.springframework.web.servlet.view.document.AbstractJExcelView (para generar archivos Excel JExcelApi) e implementar el método buildExcelDocument ().

Aquí está la lista completa para la vista POI Excel que muestra la lista de palabras desde el mapa del modelo en filas consecutivas de la primera columna de una nueva hoja de cálculo:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(Map model, HSSFWorkbook wb,
HttpServletRequest req,
            HttpServletResponse resp) throws Exception {

        HSSFSheet sheet;
        HSSFRow sheetRow;
        HSSFCell cell;

        // Go to the first sheet
        // getSheetAt: only if wb is created from an existing document
        // sheet = wb.getSheetAt(0);
        sheet = wb.createSheet("Spring");
        sheet.setDefaultColumnWidth((short) 12);

        // write a text at A1
        cell = getCell(sheet, 0, 0);
```

```
        setText(cell, "Spring-Excel test");

        List words = (List) model.get("wordList");
        for (int i=0; i < words.size(); i++) {
            cell = getCell(sheet, 2+i, 0);
            setText(cell, (String) words.get(i));
        }
    }

}
```

Y la siguiente es una vista de generar el mismo archivo de Excel, ahora usando JExcelApi:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractJExcelView {

    protected void buildExcelDocument(Map model, WritableWorkbook wb,
            HttpServletRequest request, HttpServletResponse response)
throws Exception {

        WritableSheet sheet = wb.createSheet("Spring", 0);

        sheet.addCell(new Label(0, 0, "Spring-Excel test"));

        List words = (List) model.get("wordList");
        for (int i = 0; i < words.size(); i++) {
            sheet.addCell(new Label(2+i, 0, (String) words.get(i)));
        }
    }
}
```

Hay que tener en cuenta las diferencias entre las APIs. La JExcelApi es algo más intuitiva, y además, proporciona mayor facilidad a la hora de manipular las imágenes. Sin embargo hay problemas con la memoria, al usar grandes archivos de Excel con JExcelApi.

Si se modifica el controlador para que devuelva el xl como el nombre de la vista (return new ModelAndView("xl", map);) y se ejecuta la aplicación de nuevo, se creará la hoja de cálculo Excel y se descargará automáticamente cuando se solicite la misma página que antes.

Crear subclases para las vistas PDF

La versión en PDF de la lista de palabras es aún más simple. Esta vez, la clase extiende org.springframework.web.servlet.view.document.AbstractPdfView e implementa el método buildPdfDocument () de la siguiente manera:

```
package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {
```

```
    protected void buildPdfDocument(Map model, Document doc, PdfWriter
writer,
        HttpServletRequest req, HttpServletResponse resp) throws
Exception {
        List words = (List) model.get("wordList");
        for (int i=0; i<words.size(); i++) {
            doc.add( new Paragraph((String) words.get(i)));
        }
    }

}
```

Modificando el controlador haciendo que devuelva la vista pdf con return new ModelAndView("pdf", map);, y volviendo a cargar la URL de la aplicación consegiremos un documento PDF con una lista de cada una de las palabras en el mapa.

7. JasperReports

JasperReports es un motor de informes de código abierto que apoya la creación de diseños de informes utilizando un formato fácil de entender mediante archivos XML. JasperReports es capaz de proporcionar informes en cuatro formatos diferentes: CSV, Excel, HTML y PDF.

7.1. Dependencias

La solicitud deberá incluir la última versión de JasperReports. JasperReports depende de los siguientes proyectos:

- BeanShell
- Commons BeanUtils
- Commons Colecciones
- Commons Digestor
- Commons Logging
- iText
- POI

JasperReports también requiere un analizador XML compatible con JAXP.

7.2. Configuración

Para configurar una vista JasperReports es necesario definir un ViewResolver en la configuración del contenedor de Spring para mapear nombres de vista de la clase adecuada, dependiendo del formato en el que se desee el informe.

Configuración del ViewResolver

Normalmente, se utilizará el ResourceBundleViewResolver para mapear el nombre de las vistas de las clases y los ficheros.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver
">
    <property name="basename" value="views"/>
</bean>
```

En el ejemplo anterior se ha configurado una instancia de la clase ResourceBundleViewResolver que buscará asignaciones de vistas en el paquete de recursos View.

Configuración de las Vistas

Spring Framework contiene cinco diferentes implementaciones de View para JasperReports, cuatro de los cuales corresponden a uno de los cuatro formatos de salida soportados, y uno que permite que el formato que se determine en tiempo de ejecución:

Nombre de clase	Formato de representación
JasperReportsCsvView	CSV
JasperReportsHtmlView	HTML
JasperReportsPdfView	PDF
JasperReportsXlsView	Microsoft Excel
JasperReportsMultiFormatView	La vista se decide en tiempo de ejecución

Figura 29. Clases de vistas para JasperReports.

Mapear una de estas clases para obtener un nombre de vista y un archivo de informe es cuestión de añadir las entradas apropiadas en el paquete de recursos configurados en la sección anterior, como se muestra a continuación:

simpleReport.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfVie
w
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper

Se puede ver que la vista con nombre SimpleReport se asigna a la clase JasperReportsPdfView, provocando la salida de este informe que será dictada en formato PDF. La URL característica de la vista se establece en la ubicación del archivo del informe subyacente.

Acerca de los archivos de informe

JasperReports tiene dos tipos diferentes de reportar archivos: el archivo de diseño, que tiene una extensión .jrxml, y el archivo de informe elaborado, que tiene una extensión .jaspe. Normalmente, se utiliza la tarea JasperReports Ant para compilar el archivo .jrxml en un archivo .jaspe antes de implementarlo en la aplicación. Con Spring Framework se puede asignar cualquiera de estos archivos en el archivo de informe y el framework se hará cargo de la compilación de los archivos .jrxml. Después que un archivo .jrxml sea compilado por Spring, el

informe compilado se almacena en caché para la vida útil de la aplicación. Por lo tanto, para realizar cambios en el archivo habrá que reiniciar la aplicación.

Usando JasperReportsMultiFormatView

JasperReportsMultiFormatView permite que el formato de informe se especifique en tiempo de ejecución. La representación real del informe se delega a una de las otras clases de vista de JasperReports, la clase JasperReportsMultiFormatView simplemente añade una capa que permite a la aplicación especificarse en tiempo de ejecución.

La clase JasperReportsMultiFormatView introduce dos conceptos: la clave de formato y la clave de mapeo. La clase JasperReportsMultiFormatView utiliza la clave de mapeo para buscar la vista real de la clase de implementación, y utiliza la clave de formato para buscar la clave de mapeo. Desde la perspectiva de la codificación se agrega una entrada al modelo con la clave de formato como la clave y la clave de mapeo como el valor, por ejemplo:

```
public ModelAndView handleSimpleReportMulti(HttpServletRequest
request,
HttpServletResponse response) throws Exception {

    String uri = request.getRequestURI();
    String format = uri.substring(uri.lastIndexOf(".") + 1);

    Map model = getModel();
    model.put("format", format);

    return new ModelAndView("simpleReportMulti", model);
}
```

En este ejemplo, la clave de mapeo se determina a partir de la extensión de la petición URI y se añade al modelo bajo la clave de formato por defecto: format. Si se desea utilizar una clave de formato diferente, se puede modificar mediante la propiedad formatKey de la clase JasperReportsMultiFormatView.

Por defecto las siguientes claves de mapeo se configuran en JasperReportsMultiFormatView:

Key Mapping	Clase
csv	JasperReportsCsvView
HTML	JasperReportsHtmlView
pdf	JasperReportsPdfView
xls	JasperReportsXlsView

Figura 30. Clases de mapeo de JasperReportsMultiFormarView.

Así, en el ejemplo anterior una solicitud URI/foo/myReport.pdf sería asignada a la clase JasperReportsPdfView. Se puede anular el mapeo de claves para ver el mapeo de clases utilizando la propiedad formatMappings de JasperReportsMultiFormatView.

7.3. Rellenar el ModelAndView

Para hacer correctamente un informe en el formato elegido, se deben proporcionar todos los datos necesarios para rellenar el informe. Para JasperReports esto significa que se debe rellenar todos los parámetros del informe junto con el origen de datos del informe. Los parámetros de informe son simples parejas nombre/valor y se pueden agregar al mapa del modelo, de modo que se agregaría cualquier par de nombre/valor.

Existen dos enfoques para añadir la fuente de datos al modelo, el primer método consiste en agregar una instancia del JRDataSource o una colección de tipos para el modelo de mapa bajo cualquier clave arbitraria. Entonces Spring localiza este objeto en el modelo y lo trata como el origen de datos del informe. Por ejemplo, es posible rellenar el modelo de este modo:

```
private Map getModel() {
    Map model = new HashMap();
    Collection beanData = getBeanData();
    model.put("myBeanData", beanData);
    return model;
}
```

El segundo enfoque consiste en añadir una instancia de JRDataSource o una colección bajo una clave específica y luego configurar esta clave con la propiedad reportDataKey de la vista. En ambos casos, Spring ajustará las instancias de las colecciones a una instancia JRBeanCollectionDataSource. Por ejemplo:

```
private Map getModel() {
    Map model = new HashMap();
    Collection beanData = getBeanData();
    Collection someData = getSomeData();
    model.put("myBeanData", beanData);
    model.put("someData", someData);
    return model;
}
```

Aquí se puede ver como dos instancias de colecciones se añaden al modelo. Para asegurarnos que se usan correctamente, simplemente hay que modificar la configuración de la vista:

simpleReport.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfVie
w
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
simpleReport.reportDataKey=myBeanData

Hay que tener en cuenta que cuando se utiliza el primer enfoque, Spring utilizará la primera instancia de JRDataSource o de colecciones que encuentre. Si se tiene que colocar varias instancias de JRDataSource o de colecciones en el modelo se tiene que utilizar el segundo enfoque.

7.4 Trabajar con sub-informes

JasperReports proporciona soporte para incluir sub-informes dentro de sus informes principales. Hay una amplia variedad de mecanismos para la inclusión de sub-informes. La forma más fácil es codificar la ruta del informe y la consulta SQL para el sub-informe en sus archivos de diseño. La desventaja de este enfoque es obvia: los valores están codificados de forma rígida en sus archivos de informes, lo que reduce la reutilización y produce que sea más difícil modificar los diseños de los informes.

Configuración de archivos de sub-informes

Para controlar qué los archivos de sub-informes se incluyen en un informe principal mediante Spring, el archivo de informes debe estar configurado para aceptar sub-informes de una fuente externa. Para ello se declara un parámetro en el archivo de informes de este modo:

```
<parameter name="ProductsSubReport"
class="net.sf.jasperreports.engine.JasperReport"/>
```

A continuación, se define el sub-reporte a utilizar mediante el parámetro subreporter:

```
<subreport>
    <reportElement isPrintRepeatedValues="false" x="5" y="25"
width="325"
        height="20" isRemoveLineWhenBlank="true" backcolor="#ffcc99"/>
    <subreportParameter name="City">
<subreportParameterExpression><![CDATA[$F{city}]]></subreportParameter
Expression>
    </subreportParameter>
<dataSourceExpression><![CDATA[$P{SubReportData}]]></dataSourceExpress
ion>
    <subreportExpression
class="net.sf.jasperreports.engine.JasperReport">
        <![CDATA[$P{ProductsSubReport}]]></subreportExpression>
</subreport>
```

Esto define un archivo de informe principal que espera que un sub-informe sea pasado como una instancia de net.sf.jasperreports.engine.JasperReports bajo el parámetro ProductsSubReport. Cuando se configura la clase de la vista Jasper, se puede indicar que Spring cargue un archivo de informe para que sea pasado al motor JasperReports como un sub-informe con la propiedad subReportUrls:

```
<property name="subReportUrls">
    <map>
        <entry key="ProductsSubReport" value="/WEB-
INF/reports/subReportChild.jrxml"/>
    </map>
</property>
```

Aquí, la clave del mapa se corresponde con el nombre del parámetro del sub-reporte en el archivo de informes, y la entrada es la dirección URL del archivo de informes. Spring cargará este archivo de informes, lo compilara si fuera necesario y lo pasará al motor JasperReports bajo la clave dada.

8. Vistas Feed

Tanto AbstractAtomFeedView y AbstractRssFeedView heredan de la clase base AbstractFeedView y se utilizan para proporcionar vistas de alimentación Atom y RSS. Se basan en el proyecto java.net 's ROMA y se encuentran en el paquete org.springframework.web.servlet.view.feed.

AbstractAtomFeedView es requerido para implementar el método buildFeedEntries() y, opcionalmente, para sobrescribir el método buildFeedMetadata() como se muestra a continuación.

```
public class SampleContentAtomView extends AbstractAtomFeedView {

    @Override
    protected void buildFeedMetadata(Map<String, Object> model,
            Feed feed, HttpServletRequest request) {
        // implementation omitted
    }

    @Override
    protected List<Entry> buildFeedEntries(Map<String, Object> model,
            HttpServletRequest request, HttpServletResponse response)
throws Exception {
        // implementation omitted
    }

}
```

Algo similar se aplica a la implementación AbstractRssFeedView, como se muestra a continuación.

```
public class SampleContentAtomView extends AbstractRssFeedView {

    @Override
    protected void buildFeedMetadata(Map<String, Object> model,
            Channel feed, HttpServletRequest request) {
        // implementation omitted
    }

    @Override
    protected List<Item> buildFeedItems(Map<String, Object> model,
            HttpServletRequest request, HttpServletResponse response)
throws Exception {
        // implementation omitted
    }
```

```
}
```

Los métodos buildFeedItems () y buildFeedEntires () son pasados en la petición HTTP en caso de tener que acceder a la configuración regional. La respuesta HTTP se pasa sólo por las cookies u otras cabeceras HTTP. El contenido se escribirá automáticamente en el objeto de respuesta después de ser devuelta por el método.

9. Vistas JSON Mapping

MappingJackson2JsonView (o MappingJacksonJsonView, dependiendo de la versión de Jackson) utiliza la biblioteca Jackson ObjectMapper para transformar el contenido de la respuesta a un JSON. Por defecto, todo el contenido del mapa del modelo, con la excepción de las clases específicas del framework, será codificado como un JSON. Para los casos en los que el contenido del mapa tiene que ser filtrado, los usuarios pueden especificar un conjunto específico de atributos del modelo para ser codificados a través de la propiedad RenderedAttributes.

El mapeo de los JSON se puede personalizar según sea necesario a través del uso de las anotaciones proporcionadas por Jackson. Cuando se necesita un mayor control, se puede inyectar un ObjectMapper personalizado a través de la propiedad ObjectMapper para los casos en que los JSON personalizados necesitan ser proporcionados para tipos específicos.

SPRING PORTLET MVC

1. Introducción

Además de apoyar el desarrollo Web convencional (basado en servlet), Spring también es compatible con JSR-168 para el desarrollo de portlets. En la medida de lo posible, el framework de portlets MVC es una imagen especular del framework Web MVC, y también utiliza las mismas abstracciones de vistas y las mismas tecnologías de integración.

La principal diferencia entre el flujo de trabajo de los portlets y el flujo de trabajo de los servlets es que la petición del portlet puede tener dos fases distintas: la fase action y la fase render. La fase de action se ejecuta sólo una vez, y es cuando se produce cualquier cambio en el back-end u ocurre alguna acción, como por ejemplo hacer cambios en una base de datos. La fase render se produce cuando lo que se muestra algún tipo de información al usuario, por ejemplo cada vez que se actualice la pantalla. Un punto crítico es que para una única solicitud, la fase action se ejecuta sólo una vez, pero la fase render se puede ejecutar varias veces, esto requiere una separación limpia entre las actividades que modifican el estado persistente del sistema y de las actividades que generan lo que se muestra al usuario.

Las fases duales de las peticiones de portlets son una de las fortalezas de la especificación JSR-168. Por ejemplo, los resultados de búsqueda dinámica se pueden actualizar de forma rutinaria en la pantalla sin que el usuario vuelva a realizar de forma explícita la búsqueda. La mayoría de frameworks MVC de portlets intentan ocultar completamente las dos fases del desarrollador y hacer que se vea como el desarrollo de servlets tradicional.

La manifestación principal de este enfoque es que las clases MVC de la versión servlet tendrán un método que se ocupe de la solicitud, mientras que la versión de portlet de las clases MVC tendrá dos métodos que tienen que ver con la solicitud, uno para la fase action y una para la fase render. Por ejemplo, mientras que la versión de AbstractController para servlets tiene el método handleRequestInternal (..) , la versión de AbstractController para portlets tiene los métodos handleActionRequestInternal (..) y handleRenderRequestInternal (..).

El framework está diseñado en torno a un DispatcherPortlet que envía solicitudes a los controladores, con mapeos del controlador configurables y resolución de vistas.

La resolución regional y la resolución del tema no son admitidas en portlets MVC, sin embargo, todos los mecanismos disponibles en Spring que dependen de la configuración regional (como la internacionalización de los mensajes) seguirán funcionando correctamente porque el DispatcherPortlet expone la localización actual de la misma manera que lo hace el DispatcherServlet.

1.1. Los controladores

El controlador predeterminado sigue siendo un controlador de interfaz muy simple, que ofrece sólo dos métodos:

- void handleActionRequest(request,response)
- ModelAndView handleRenderRequest(request,response)

El framework también incluye, en su mayor parte, la misma jerarquía de aplicación del controlador, tal como AbstractController, SimpleFormController, etc. El enlace de datos, el uso de objetos command, el modelo de control y la resolución de vistas son iguales que en el framework para servlets.

1.2. Las vistas

Todas las capacidades de representación de vistas de los servlets se utilizan directamente a través de un servlet puente especial llamado ViewRendererServlet .Mediante el uso de este servlet, la solicitud portlet se convierte en una petición de servlet y la vista se puede representar usando toda la infraestructura normal para servlets. Esto significa que todos los procesadores existentes, como JSP, Velocity, etc, se pueden seguir utilizando para los portlets.

1.3. Los beans

Spring Portlet MVC soporta beans cuyo ciclo de vida estén en el ámbito de la solicitud HTTP o sesión HTTP actuales. Esto en sí, no es una característica específica de Spring Portlets MVC, sino más bien del recipiente WebApplicationContext que Spring utiliza.

2. El DispatcherPortlet

Portlets MVC es un framework Web MVC, diseñado en torno a un portlet que envía solicitudes a los controladores y que ofrece otras funcionalidades para facilitar el desarrollo de aplicaciones de portlets. Spring DispatcherPortlet sin embargo, no se limita a eso. Está completamente integrado con el Spring ApplicationContext, lo que le permite utilizar todas las demás característica que proporciona Spring.

Como portlets ordinarios, el DispatcherPortlet se declara en el archivo portlet.xml de la aplicación web:

```
<portlet>
    <portlet-name>sample</portlet-name>
    <portlet-
class>org.springframework.web.portlet.DispatcherPortlet</portlet-
class>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
        <title>Sample Portlet</title>
    </portlet-info>
</portlet>
```

En el framework para portlets MVC, cada DispatcherPortlet tiene su propio contexto de aplicación web (WebApplicationContext), que hereda todos los beans que ya estén definidos en el RootWebApplicationContext. Estos beans heredados se pueden sobrescribir en el ámbito de portlets específico, y los nuevos beans de alcance específico pueden definirse de manera local para una instancia de portlet determinada.

Para la inicialización de un DispatcherPortlet, hay que crear los beans en un archivo llamado [portlet-name]-portlet.xml en el directorio WEB-INF de la aplicación web, también hay que sobrescribir la definición de cualquier bean definido con el mismo nombre en el ámbito global.

Spring DispatcherPortlet tiene algunos beans especiales que utiliza con el fin de ser capaz de procesar las solicitudes y hacer las vistas apropiadas. Estos beans se incluyen en el framework y se pueden configurar en el WebApplicationContext, del mismo modo que se configuraría cualquier otro bean.

Cuando un DispatcherPortlet está configurado para su uso y llega una petición para un determinado DispatcherPortlet, comienza el procesamiento de la solicitud. La lista a continuación describe el proceso completo por el que pasa una solicitud si se maneja con un DispatcherPortlet:

1. La localización devuelta por PortletRequest.getLocale() se une a la petición para permitir a los elementos del proceso resolver la configuración regional que se utilizará al procesar la solicitud (la representación de la vista, la preparación de los datos, etc.)
2. Si se especifica un dispositivo de resolución multi-parte y este es un ActionRequest, la solicitud se inspecciona para Multi-partes y si se encuentran, se envuelven en un MultipartActionRequest para su posterior procesamiento por otros elementos del proceso.
3. Se busca un controlador adecuado. Si no se encuentra un controlador, la cadena de ejecución asociado con el controlador (pre-procesadores, post-procesadores, controladores) se ejecutará con el fin de elaborar un modelo.
4. Si se devuelve un modelo, la vista se representa mediante la resolución de vistas de que se ha configurado con el WebApplicationContext. Si no se devuelve ningún modelo, la vista no se representa, ya que la solicitud podría haberse cumplido ya.

Las excepciones que se producen durante el procesamiento de la solicitud son recogidas por cualquiera de los controladores de excepciones que se declaran en el WebApplicationContext. El uso de estos controladores de excepción se puede definir con un comportamiento personalizado en caso de que tales excepciones sean lanzadas.

Se puede personalizar el DispatcherPortlet agregando parámetros de contexto en el archivo portlet.xml. Las posibilidades son las siguientes:

Parámetro	Explicación
ContextClass	Clase que implementa el WebApplicationContext y que se utiliza para crear una instancia del contexto utilizado por este portlet. Si no se especifica este parámetro, se utilizará el XmlPortletApplicationContext.
ContextConfigLocation	String que se pasa a la instancia del contexto (especificado por ContextClass) para indicar donde se puede encontrar el contexto. El String se divide en varios Strings (utilizando una coma como delimitador) para soportar múltiples contextos (en el caso de utilizar distintos contextos para los beans que se definen en dos ocasiones, el último tiene prioridad).
Namespace	El espacio de nombres del WebApplicationContext. El valor predeterminado es [portlet-name]-portlet.
ViewRendererUrl	La URL en la que el DispatcherPortlet puede acceder a una instancia de ViewRendererServlet.

Figura 31. Parámetros del DispatcherPortlet.

3. El ViewRendererServlet

El proceso de renderización en portlets MVC es un poco más complejo que en servlets. Con el fin de volver a utilizar todas las tecnologías de representación de vistas de Spring Web MVC, se debe convertir el PortletRequest/PortletResponse en un HttpServletRequest/HttpServletResponse y luego llamar al método render de la vista .Para ello, DispatcherPortlet utiliza un servlet especial que existe para este propósito: el ViewRendererServlet .

Para que el DispatcherPortlet sea renderizado, se debe declarar de la siguiente manera una instancia del ViewRendererServlet en el archivo web.xml:

```
<servlet>
    <servlet-name>ViewRendererServlet</servlet-name>
    <servlet-
class>org.springframework.web.servlet.ViewRendererServlet</servlet-
class>
</servlet>

<servlet-mapping>
    <servlet-name>ViewRendererServlet</servlet-name>
    <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

Para llevar a cabo la representación real, DispatcherPortlet hace lo siguiente:

1. Enlaza el WebApplicationContext a la solicitud como un atributo con la misma clave WEB_APPLICATION_CONTEXT_ATTRIBUTE que el DispatcherServlet utiliza.
2. Asocia los objetos de los modelos y de las vistas a la solicitud para que puedan ser usados por el ViewRendererServlet.
3. Construye un PortletRequestDispatcher y realiza un include usando la URL /WEB-INF/servlet/view mapeada por el ViewRendererServlet.

ViewRendererServlet es entonces capaz de llamar al método render de la vista con los argumentos adecuados.

4. Los controladores

Los controladores de portlets MVC son muy similares a los controladores Web del MVC, lo que hace simple compartir código entre ambos.

La base para la arquitectura del controlador de portlets MVC es la interfaz org.springframework.web.portlet.mvc.Controller, que se muestra a continuación.

```
public interface Controller {

    /**
     * Process the render request and return a ModelAndView object
which the
     * DispatcherPortlet will render.
     */
    ModelAndView handleRenderRequest(RenderRequest request,
            RenderResponse response) throws Exception;

    /**
     * Process the action request. There is nothing to return.
     */
    void handleActionRequest(ActionRequest request,
            ActionResponse response) throws Exception;

}
```

Como se puede ver, la interfaz del controlador de portlet requiere dos métodos que controlan las dos fases de una solicitud portlet: La petición action y la petición render. La fase action debe ser capaz de manejar una solicitud action, y la fase render, debe ser capaz de manejar una solicitud render, y devolver un modelo y una vista apropiados. Mientras que la interfaz del controlador es bastante abstracta, Spring MVC portlet ofrece varios controladores que ya contienen una gran cantidad de la funcionalidad que pueden ser usados, siendo la mayoría de ellos muy similares a los controladores del Spring Web MVC. La interfaz del controlador sólo define la funcionalidad requerida por cada controlador: el manejo de una solicitud action, el manejo de una solicitud render y la devolución de un modelo y de una vista.

4.1. AbstractController y PortletContentGenerator

Por supuesto, sólo una interfaz para el controlador no es suficiente. Para proporcionar una infraestructura básica, todos los controladores Spring portlets MVC heredan de AbstractController, lo que proporciona acceso al ApplicationContext y control sobre el almacenamiento en la memoria caché.

A continuación se muestran las características que proporciona AbstractController:

Parámetro	**Explicación**
requireSession	Indica si este controlador requiere una sesión para hacer su trabajo. Esta función se ofrece a todos los controladores. Si una sesión no está presente cuando un controlador de este tipo recibe una solicitud, se informa al usuario utilizando una SessionRequiredException.
synchronizeSession	El controlador anulará los métodos handleRenderRequestInternal(..) y handleActionRequestInternal (..), que se sincronizarán en la sesión del usuario si especifica esta variable.
renderWhenMinimized	Si se desea que el controlador muestre la vista cuando el portlet se encuentre minimizado, hay que poner este parámetro a true. De forma predeterminada, se establece como false para que los portlets que se encuentran en un estado minimizado no muestren ningún contenido.
cacheSeconds	Cuando se desea que un controlador anule la caducidad de la caché definida por el portlet, es necesario especificar un entero positivo. Por defecto se establece en -1, lo que no cambia el almacenamiento en caché de forma predeterminada. Si se establece como 0 se asegurará que el resultado nunca se almacene en caché.

Figura 32. Características de AbstractController.

Las propiedades requireSession y cacheSeconds se declaran en la clase PortletContentGenerator, que es la superclase de AbstractController.

Cuando se utiliza el AbstractController como una clase base únicamente hay que sobrescribir el método handleActionRequestInternal (ActionRequest, ActionResponse), el método handleRenderRequestInternal (RenderRequest, renderResponse) o ambos, aplicar la lógica, y devolver un objeto ModelAndView en el caso de handleRenderRequestInternal.

Las implementaciones predeterminadas de ambos handleActionRequestInternal(..) y handleRenderRequestInternal(..) arrojan una PortletException . Esto es consistente con el comportamiento de GenericPortlet a partir de la especificación API JSR-168. Por lo que sólo es necesario reemplazar el método que el controlador tiene la intención de manejar.

Aquí se muestra un ejemplo breve formado por una clase y una declaración en el contexto de la aplicación web.

```
package samples;

import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.springframework.web.portlet.mvc.AbstractController;
import org.springframework.web.portlet.ModelAndView;

public class SampleController extends AbstractController {

    public ModelAndView handleRenderRequestInternal(RenderRequest
request, RenderResponse response) {
        ModelAndView mav = new ModelAndView("foo");
        mav.addObject("message", "Hello World!");
        return mav;
    }

}
```

```
<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/>
</bean>
```

La clase anterior y la declaración en el contexto de aplicaciones web, es todo lo que se necesita, además de la creación de un mapeo para el controlador para obtener un controlador.

4.2. Otros controladores

Aunque se puede extender AbstractController, Spring MVC portlet proporciona una serie de implementaciones concretas que ofrecen funcionalidades que se utilizan comúnmente en aplicaciones MVC.

El ParameterizableViewController es básicamente el mismo que el usado en el ejemplo anterior, excepto por el hecho de que se puede especificar el nombre de la vista que va ser devuelta, en el contexto de aplicaciones web (sin necesidad de codificar el nombre de la vista).

El PortletModeNameViewController utiliza el modo actual del portlet como el nombre de la vista. Por lo tanto, si su portlet está en modo View (es decir, PortletMode.VIEW) entonces utiliza "view" como el nombre de la vista.

4.3. Controladores de comandos

Spring portlet MVC tiene exactamente la misma jerarquía para los controladores de comando que Spring Web MVC. Proporcionan una forma de interactuar con los objetos y enlazar dinámicamente parámetros del PortletRequest al objeto especificado. Sus objetos no tienen que implementar una interfaz específica, por lo que pueden manipular directamente los objetos persistentes. Se va a examinar que comandos están disponibles en los controladores, para obtener una visión general de lo que se puede hacer con ellos:

- AbstractCommandController: un controlador de comandos que se puede utilizar para crear un controlador del sistema propio, capaz de unirse a los parámetros de solicitud de un objeto que especifique. Esta clase no ofrece la funcionalidad de formularios, sin embargo ofrece funciones de validación y permite especificar en el controlador qué hacer con el objeto de comando que se ha llenado con los parámetros de la petición.
- AbstractFormController: un controlador abstracto de apoyo a la presentación de formularios. El uso de este controlador permite modelar los formularios y rellenarlos con un objeto de comando. Después de que un usuario haya llenado el formulario, AbstractFormController une los campos, valida y entrega el objeto al controlador para tomar las medidas adecuadas.
- SimpleFormController: un AbstractFormController concreto que proporciona apoyo al crear un formulario con un objeto de comando correspondiente. SimpleFormController permite especificar entre otras cosas un objeto de comando, un nombre de vista para el formulario, un nombre de vista para la página que desea mostrar al usuario cuando el envío de formularios ha tenido éxito.
- AbstractWizardFormController: un AbstractFormController concreto que proporciona una interfaz para editar el contenido de un objeto de comando a través de múltiples páginas de visualización. Soporta múltiples acciones de los usuarios: acabado, cancelar o cambiar la página, todos los cuales son fácilmente especificados en la solicitud de parámetros desde la vista.

Estos controladores de comandos son muy poderosos, pero requieren un conocimiento detallado de la forma en la que operan con el fin de utilizarlos de manera eficiente.

4.4. PortletWrappingController

En lugar de desarrollar nuevos controladores, es posible utilizar los portlets y las peticiones del mapa existentes desde un DispatcherPortlet. Usando elPortletWrappingController, se pueden crear instancias de un portlet existente como un controlador de la siguiente manera:

```
<bean id="myPortlet"
class="org.springframework.web.portlet.mvc.PortletWrappingController">
    <property name="portletClass" value="sample.MyPortlet"/>
    <property name="portletName" value="my-portlet"/>
    <property name="initParameters">
        <value>config=/WEB-INF/my-portlet-config.xml</value>
    </property>
</bean>
```

Esto puede ser muy valioso, ya que se puede utilizar interceptores de pre y post proceso de las peticiones que van a estos portlets. Desde JSR-168 no se admite ningún tipo de mecanismo de filtro, y esto es bastante práctico. Por ejemplo, esto puede ser utilizado para envolver el Hibernate OpenSessionInViewInterceptor alrededor de un MyFaces JSF de portlets.

5. Mapeo del controlador

El uso del mapeo del controlador permite asignar las solicitudes de portlets entrantes a los controladores apropiados. La funcionalidad básica de un HandlerMapping ofrece un HandlerExecutionChain, que debe contener al controlador que coincide con la petición de entrada, y también puede contener una lista de interceptores del controlador que se aplican a la solicitud. Cuando llega una petición, el DispatcherPortlet lo entregará al mapa del controlador para inspeccionar la solicitud y llegar a un HandlerExecutionChain adecuado. Entonces el DispatcherPortlet ejecutará el controlador y los interceptores.

En Spring Web MVC, el mapeo del controlador se basa comúnmente en URLs. Dado que en realidad no hay tal cosa como una URL dentro de un portlet, debemos usar otros mecanismos para controlar los mapeos. Los dos más comunes son el modo portlet y un parámetro de petición, pero ninguno se puede utilizar con un controlador personalizado.

En el resto de esta sección se describen tres de los mapeos más utilizados. Todos ellos heredan de AbstractHandlerMapping y comparten las siguientes propiedades:

- interceptors: La lista de los interceptores que se utilizan.
- DefaultHandler: El controlador predeterminado cuando esta asignación de controlador no da lugar a un controlador coincidente.
- order: Basado en el valor de la propiedad de order, Spring clasificará todas las asignaciones de controlador disponibles en el contexto y aplicará el primer controlador coincidente.
- lazyInitHandlers: Permite la inicialización de los manipuladores simples. El valor predeterminado es falso. Esta propiedad se implementa directamente en los tres manipuladores concretos.

5.1. PortletModeHandlerMapping

Se trata de un simple mapeo de controlador que asigna las peticiones entrantes en función de la modalidad actual del portlet. Un ejemplo:

```
<bean
class="org.springframework.web.portlet.handler.PortletModeHandlerMappi
ng">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="viewHandler"/>
```

```
            <entry key="edit" value-ref="editHandler"/>
            <entry key="help" value-ref="helpHandler"/>
        </map>
    </property>
</bean>
```

5.2. ParameterHandlerMapping

Si se tiene que navegar alrededor de varios controladores sin necesidad de cambiar el modo de portlet, la forma más sencilla de hacerlo es con un parámetro de la petición que se utiliza como la clave para el control del mapeo.

ParameterHandlerMapping utiliza el valor de un parámetro específico de la petición para controlar el mapeo. El nombre por defecto del parámetro es 'action', pero se puede cambiar con la propiedad 'parameterName'.

La configuración del bean para este mapeo será algo como esto:

```
<bean
class="org.springframework.web.portlet.handler.ParameterHandlerMapping
">
    <property name="parameterMap">
        <map>
            <entry key="add" value-ref="addItemHandler"/>
            <entry key="edit" value-ref="editItemHandler"/>
            <entry key="delete" value-ref="deleteItemHandler"/>
        </map>
    </property>
</bean>
```

5.3. PortletModeParameterHandlerMapping

La capacidad más potente de PortletModeParameterHandlerMapping combina las capacidades de los dos anteriores para permitir una navegación diferente dentro de cada modo portlet.

Una vez más el nombre por defecto del parámetro es "action", pero se puede cambiar con la propiedad ‘parameterName’.

Por defecto, el valor del parámetro no se puede utilizar en dos modos de portlets diferentes. Esto es para que si el propio portal cambia el modo, la solicitud ya no será válida en el mapeo. Este comportamiento se puede cambiar estableciendo la propiedad allowDupParameters como true. Sin embargo, no es recomendable.

La configuración del bean para este mapeo será parecido a esto:

```
<bean
class="org.springframework.web.portlet.handler.PortletModeParameterHan
dlerMapping">
    <property name="portletModeParameterMap">
        <map>
            <entry key="view"> <!-- view portlet mode -->
                <map>
                    <entry key="add" value-ref="addItemHandler"/>
                    <entry key="edit" value-ref="editItemHandler"/>
                    <entry key="delete" value-
ref="deleteItemHandler"/>
                </map>
            </entry>
            <entry key="edit"> <!-- edit portlet mode -->
                <map>
                    <entry key="prefs" value-ref="prefsHandler"/>
                    <entry key="resetPrefs" value-
ref="resetPrefsHandler"/>
                </map>
            </entry>
        </map>
    </property>
</bean>
```

5.4. Añadiendo HandlerInterceptors

Los mecanismos de mapeo de controladores de Spring tienen nociones de interceptores de controlador, lo que resulta muy útil cuando se desea aplicar una funcionalidad específica a ciertas peticiones. Una vez más Spring Portlet MVC implementa estos conceptos de la misma manera que Spring Web MVC.

Los interceptores ubicados en el mapeo del controlador deben implementar HandlerInterceptor del paquete org.springframework.web.portlet. Al igual que la versión de servlet, esta interfaz define tres métodos: uno que se llamará antes de que se ejecute el controlador actual (preHandle), uno que se llamará después de que se ejecute el controlador (postHandle), y uno que se llama después que la solicitud haya terminado (afterCompletion). Estos tres métodos deben proporcionar la flexibilidad suficiente para hacer todo tipo de pre-y post-procesamiento.

El método preHandle devuelve un valor booleano. Se puede utilizar este método para detener o continuar con el procesamiento de la cadena de ejecución. Cuando este método devuelve true, continuará la cadena de ejecución del gestor. Cuando devuelve false, el DispatcherPortlet asume que el interceptor se ha encargado de las peticiones y detiene la ejecución de los otros interceptores y del controlador de la cadena de ejecución.

El método postHandle sólo se llama en un RenderRequest. Los métodos preHandle y afterCompletion son llamados a la vez mediante un ActionRequest y un RenderRequest.

6. Vistas y resolutores de vista

Como se mencionó anteriormente, Spring MVC portlet reutiliza directamente todas las tecnologías de vista del Spring Web MVC. Esto incluye no sólo las implementaciones de Vistas propias, sino también las implementaciones de los ViewResolver.

Unos pocos items que usan las Vistas existentes y las implementaciones de los ViewResolver son dignos de mención:

- La mayoría de los portales esperan que el resultado de la prestación de un portlet sea un fragmento de código HTML. Así, las vistas como JSP/JSTL, Velocity, FreeMarker y XSLT pueden utilizarse.
- No hay redirección HTTP desde un portlet (el método sendRedirect(..) de ActionResponse no se puede utilizar para mantenerse dentro del portal). Así, RedirectView y el uso del prefijo "redirect:" no funciona correctamente desde portlets MVC.
- Puede que sea posible utilizar el prefijo 'fordward: " dentro de portlets MVC. Sin embargo, hay que recordar que en un portlet, no se conoce la dirección URL actual. Esto significa que no se puede utilizar una dirección URL relativa para acceder a otros recursos de la aplicación web y que se tendrá que utilizar una URL absoluta.

Asimismo, para el desarrollo de JSP, las nuevas colecciones Spring Taglib y Spring Form Taglib trabajan en las vistas de portlets exactamente de la misma forma en que trabajan en las vistas de servlets.

7. Carga de archivos Multipartes

Spring Portlet MVC tiene soporte para la carga de archivos multiparte en aplicaciones de portlets. El diseño para el apoyo se realiza con los objetos PortletMultipartResolver, definidos en el paquete org.springframework.web.portlet.multipart. Spring proporciona un PortletMultipartResolver para su uso con Commons FileUpload.

Por defecto, no hay un manejador multiparte en Spring Portlets MVC, ya que algunos desarrolladores querrán manejar las multipartes por sí mismos, por lo que para poder usarlo habrá que añadir un dispositivo de resolución multipartes en el contexto de la aplicación web. Después de haber hecho eso, DispatcherPortlet inspeccionará cada solicitud para ver si contiene varias partes. Si no se encuentran, la solicitud continuará como se esperaba. Sin embargo, si se encuentran varias partes en la solicitud, el PortletMultipartResolver que se ha declarado en el contexto será utilizado.

7.1. PortletMultipartResolver

El siguiente ejemplo muestra cómo utilizar el CommonsPortletMultipartResolver:

```
<bean id="portletMultipartResolver"

class="org.springframework.web.portlet.multipart.CommonsPortletMultipa
rtResolver">
    <!-- one of the properties available; the maximum file size in
bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

Por supuesto, también hay que poner los .jar apropiados en el classpath para que la resolución de multipartes trabaje correctamente. En el caso del CommonsMultipartResolver, es necesario utilizar commons-fileupload.jar.

Cuando DispatcherPortlet detecta una solicitud multiparte, se activa el sistema de resolución que se ha declarado en su contexto. Entonces, lo que el resolutor hace, es envolver el actual ActionRequest en un MultipartActionRequest que tiene soporte para la carga de archivos multipartes. Usando el MultipartActionRequest se puede obtener información acerca de las partes contenidas por esta solicitud y obtener realmente acceso a los mismos archivos con copias de los controladores.

7.2. Manejo de un archivo en un formulario

Después que el PortletMultipartResolver ha terminado de hacer su trabajo, la solicitud será procesada como cualquier otra. Para utilizar el PortletMultipartResolver hay que crear un formulario con un campo de subida (ver ejemplo de abajo), entonces Spring se une al archivo en su formulario. Para que realmente el usuario suba un archivo, tenemos que crear un formulario (JSP / HTML):

```
<h1>Please upload a file</h1>
<form method="post" action="<portlet:actionURL/>"
enctype="multipart/form-data">
    <input type="file" name="file"/>
    <input type="submit"/>
</form>
```

Como se puede ver, se ha creado un campo denominado "file", que coincide con la propiedad del bean que tiene el array byte[]. Además se ha añadido el atributo de codificación (enctype="multipart/form-data"), que es necesario dejar para que el navegador sepa cómo codificar los campos de varias partes.

Para poder poner los datos binarios en los objetos que hay que registrar un editor personalizado con el PortletRequestDataBinder. Hay un par de editores disponibles para el manejo de archivos y establecer los resultados en un objeto. Hay un StringMultipartFileEditor

capaz de convertir archivos a Strings (utilizando un juego de caracteres definido por el usuario), y hay un ByteArrayMultipartFileEditor que convierte los archivos en matrices de bytes. Ellos función análogamente al CustomDateEditor.

Por lo tanto, para poder cargar archivos con un formulario, hay que declarar la resolución, un mapeo a un controlador que va a procesar el bean, y el propio controlador.

```
<bean id="portletMultipartResolver"

class="org.springframework.web.portlet.multipart.CommonsPortletMultipa
rtResolver"/>

<bean
class="org.springframework.web.portlet.handler.PortletModeHandlerMappi
ng">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="fileUploadController"/>
        </map>
    </property>
</bean>

<bean id="fileUploadController" class="examples.FileUploadController">
    <property name="commandClass" value="examples.FileUploadBean"/>
    <property name="formView" value="fileuploadform"/>
    <property name="successView" value="confirmation"/>
</bean>
```

Después de eso, se crea el controlador y la clase para mantener la propiedad de archivos.

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(ActionRequest request, ActionResponse
response,
            Object command, BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        byte[] file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }

    protected void initBinder(PortletRequest request,
            PortletRequestDataBinder binder) throws Exception {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor
        binder.registerCustomEditor(byte[].class, new
ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert
    }
```

```
}

public class FileUploadBean {

    private byte[] file;

    public void setFile(byte[] file) {
        this.file = file;
    }

    public byte[] getFile() {
        return file;
    }

}
```

Como se puede ver, el FileUploadBean tiene una propiedad de tipo byte [] que contiene el archivo. El controlador registra un editor personalizado para dejar que Spring sepa cómo convertir los objetos multipartes que el resolver ha encontrado en las propiedades especificadas por el bean. En este ejemplo, no se hace nada con la propiedad byte[] del propio bean, pero en la práctica se puede hacer lo que se quiera (guardarla en una base de datos, enviarla por correo, etc.)

Un ejemplo equivalente en el que se vincula un archivo directamente a una propiedad String-typed de un objeto podría tener este aspecto:

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(ActionRequest request, ActionResponse
response,
            Object command, BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        String file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }

    protected void initBinder(PortletRequest request,
            PortletRequestDataBinder binder) throws Exception {

        // to actually be able to convert Multipart instance to a
String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class, new
StringMultipartFileEditor());
        // now Spring knows how to handle multipart objects and
convert
    }
}
```

```
public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

Otra opción es unir directamente una propiedad MultipartFile declarada en la clase del objeto. En este caso no se necesita el registro de cualquier editor de propiedades personalizado porque no hay una conversión de tipos que realizar.

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(ActionRequest request, ActionResponse
response,
            Object command, BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        MultipartFile file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }

}
```

8. Manejo de excepciones

Al igual que Servlet MVC, Portlets MVC proporciona HandlerExceptionResolvers para manejar las excepciones inesperadas que se producen mientras una solicitud está siendo procesada por un controlador. Portlets MVC también proporciona un resolutor específico, SimpleMappingExceptionResolver, que le permite tomar el nombre de cualquier clase de excepción que se pueda dar y asignarla a un nombre de vista.

9. Anotaciones para la configuración del controlador

En Spring 2.5 se introdujo un modelo de programación basado en anotaciones para configurar los controladores MVC, estas anotaciones son @RequestMapping, @RequestParam, @ModelAttribute, etc. Estas anotaciones están disponibles tanto para Servlet MVC como para Portlets MVC. Los controladores implementados en este estilo no tienen que extender clases específicas o implementar interfaces específicas. Además, por lo general no tienen dependencias directas con la Api de Servlets o Portlets, a pesar de que puede obtener fácilmente acceso a las instalaciones de Servlet o Portlet.

Las siguientes secciones documentan estas anotaciones y cómo se utilizan con mayor frecuencia en un entorno de portlets.

9.1. Configuración del dispatcher

@RequestMapping sólo será procesado si el correspondiente HandlerMapping (para anotaciones a nivel de tipo) y/o HandlerAdapter (para anotaciones a nivel de método) están presentes en el dispatcher.

Sin embargo, si se definen HandlerMappings o HandlerAdapters personalizados, entonces es necesario asegurarse que se ha definido un correspondiente un DefaultAnnotationHandlerMapping y/o un AnnotationMethodHandlerAdapter personalizados, siempre y cuando se tenga la intención de utilizar @RequestMapping.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean
class="org.springframework.web.portlet.mvc.annotation.DefaultAnnotatio
nHandlerMapping"/>

    <bean
class="org.springframework.web.portlet.mvc.annotation.AnnotationMethod
HandlerAdapter"/>

    // ... (controller bean definitions) ...
```

```
</beans>
```

9.2. Definición de un controlador con @Controller

La anotación @Controller indica que una clase particular hace el papel de un controlador. No hay necesidad de extender ninguna clase base de controlador o hacer referencia a la API de Portlets. Además, se sigue siendo capaz de hacer referencia a las características de Portlet específico.

El propósito básico de la anotación @Controller es actuar como un estereotipo para la clase anotada, lo que indica su papel. El dispatcher explorará estas clases anotadas con los métodos de mapeo, detectando anotaciones @RequestMapping.

Los beans anotados del controlador se pueden definir de forma explícita, usando la definición de beans estándar de Spring en el contexto del dispatcher. Sin embargo, el estereotipo @Controller también permite la detección automática, en línea con el soporte general de Spring 2.5 para la detección de componentes de clases en la ruta de clase y las definiciones de beans para auto-registrarse.

Para habilitar la detección automática de dichos controladores anotados, hay que añadir el componente de exploración a la configuración. Esto se logra fácilmente mediante el uso del esquema de contexto de Spring, tal y como se muestra en el siguiente fragmento de código XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-
context.xsd">

    <context:component-scan base-
package="org.springframework.samples.petportal.portlet"/>

    // ...

</beans>
```

9.3. Asignación de peticiones con @RequestMapping

La anotación @RequestMapping se utiliza para mapear los modos de los Portlets como VIEW/EDIT en una clase o en un método de controlador. Normalmente la anotación a nivel de tipo mapea un modo específico en un controlador de formulario.

El siguiente es un ejemplo de un controlador de formulario de la aplicación de ejemplo PetPortal en la que se muestra el uso de esta anotación:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {

    private Properties petSites;

    public void setPetSites(Properties petSites) {
        this.petSites = petSites;
    }

    @ModelAttribute("petSites")
    public Properties getPetSites() {
        return this.petSites;
    }

    @RequestMapping // default (action=list)
    public String showPetSites() {
        return "petSitesEdit";
    }

    @RequestMapping(params = "action=add") // render phase
    public String showSiteForm(Model model) {
        // Used for the initial form as well as for redisplaying with
errors.
        if (!model.containsAttribute("site")) {
            model.addAttribute("site", new PetSite());
        }

        return "petSitesAdd";
    }

    @RequestMapping(params = "action=add") // action phase
    public void populateSite(@ModelAttribute("site") PetSite petSite,
            BindingResult result, SessionStatus status, ActionResponse
response) {
        new PetSiteValidator().validate(petSite, result);
        if (!result.hasErrors()) {
            this.petSites.put(petSite.getName(), petSite.getUrl());
            status.setComplete();
            response.setRenderParameter("action", "list");
        }
    }

    @RequestMapping(params = "action=delete")
    public void removeSite(@RequestParam("site") String site,
ActionResponse response) {
        this.petSites.remove(site);
        response.setRenderParameter("action", "list");
    }}
```

9.4. Argumentos admitidos del método controlador

Los métodos de controlador que están anotados con @RequestMapping se les permite tener llamadas muy flexibles. Pueden tener argumentos de los siguientes tipos, en orden arbitrario (con excepción de los resultados de validación, que deben estar a continuación del objeto comando correspondiente):

- Objetos de solicitud y/o de respuesta (Portlets de API). Se puede elegir cualquier tipo de petición/respuesta específica, por ejemplo PortletRequest/ActionRequest/ RenderRequest. Una declaración explicita del argumento action/render también se utiliza para el mapeo específico de los tipos de peticiones de un método de control.
- Objeto Session (API de portlets): de tipo PortletSession. Un argumento de este tipo reforzará la presencia de la sesión correspondiente. Como consecuencia, este argumento nunca será nulo.
- org.springframework.web.context.request.WebRequest u org.springframework.web.context.request.NativeWebRequest. Permite el acceso al parámetro de la petición, así como acceso a los atributos de petición/sesión, sin vínculos con la API Servlet/Portlets.
- java.util.Locale de la solicitud actual configuración regional.
- java.util.TimeZone/java.time.ZoneId para la zona horaria actual petición.
- java.io.InputStream/java.io.Reader para acceder al contenido de la solicitud.
- java.io.OutputStream/java.io.Writer para generar el contenido de la respuesta.
- La anotación @RequestParam de parámetros para el acceso a los parámetros de la petición de portlets específicos.
- java.util.Map/org.springframework.ui.Model/org.springframework.ui.ModelMap para enriquecer el modelo implícito a la que vaya a estar expuesta la Vista.
- Objetos comando/formulario para enlazar parámetros: como propiedades de beans o campos, con la conversión de tipos personalizados, dependiendo de los métodos @InitBinder y/o la configuración. Tales objetos comando, junto con sus resultados de validación se expondrán como atributos del modelo, por defecto utilizando el nombre de clase de comando no calificada en la notación de la propiedad (por ejemplo, "orderAddress" para el tipo "mypackage.OrderAddress").
- org.springframework.validation.Errors/org.springframework.validation.BindingResult resultado de la validación de un objeto comando/formulario anterior (el argumento inmediato anterior).
- org.springframework.web.bind.support.SessionStatus estado para el procesamiento de formularios.

Los siguientes tipos de retorno son compatibles con los métodos de controlador:

- Un objeto ModelAndView, con el modelo implícitamente enriquecido con objetos de comando y los resultados de los métodos de datos anotados con @ModelAttribute.
- Un objeto Model, con el nombre de la vista determinado implícitamente a través de un RequestToViewNameTranslator y el modelo implícitamente enriquecido con objetos comando y los resultados de los métodos de datos anotados con @ModelAttribute.
- Un objeto Map para la exposición de un modelo, con el nombre de la vista determinado implícitamente a través de un RequestToViewNameTranslator y el

modelo implícitamente enriquecido con objetos comando y los resultados de los métodos de datos anotados con @ModelAttribute.

- Un objeto View, con el modelo determinado implícitamente a través de objetos comando y métodos de acceso a datos con la anotación @ModelAttribute. El método de control también se puede enriquecer mediante la programación del modelo.
- Un String que se interpreta como el nombre de vista, con el modelo determinado de forma implícita a través de objetos de comando y métodos de acceso a datos con la anotación @ModelAttribute. El método de control también se puede enriquecer mediante la programación del modelo.
- Void, si el método se encarga de la respuesta en sí misma.
- Cualquier otro tipo de devolución se considerará como un atributo del modelo que se expone a la vista, utilizando el nombre de atributo especificado a través de @ModelAttribute a nivel de. El modelo se enriquecerá de forma implícita con objetos comando y el modelo implícitamente enriquecido con objetos comando y los resultados de los métodos de datos anotados con @ModelAttribute.

9.5. Enlazando parámetros con @RequestParam

La anotación @RequestParam se utiliza para enlazar los parámetros de solicitud al parámetro de un método en el controlador.

El siguiente fragmento de código muestra su uso:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {

    // ...

    public void removeSite(@RequestParam("site") String site,
ActionResponse response) {
        this.petSites.remove(site);
        response.setRenderParameter("action", "list");
    }

    // ...

}
```

Los parámetros que utilizan esta anotación son obligatorios, pero se puede especificar que un parámetro es opcional estableciendo el atributo required a false, por ejemplo, @RequestParam (value = "id", required = false).

9.6. Enlace de datos del modelo con @ModelAttribute

@ModelAttribute tiene dos escenarios de uso en los controladores. Cuando se coloca en el parámetro de un método, @ModelAttribute, se utiliza para mapear un atributo del modelo en un método anotado. Así es como el controlador obtiene una referencia al objeto que contiene los datos introducidos en el formulario.

@ModelAttribute también se utiliza a nivel de método para proporcionar datos de referencia al modelo. Para este uso el método puede contener los mismos tipos como se ha comentado anteriormente para la anotación @RequestMapping.

El siguiente fragmento de código muestra estos dos usos:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {

    // ...

    @ModelAttribute("petSites")
    public Properties getPetSites() {
        return this.petSites;
    }

    @RequestMapping(params = "action=add") // action phase
    public void populateSite( @ModelAttribute("site") PetSite petSite,
BindingResult result, SessionStatus status, ActionResponse response) {
        new PetSiteValidator().validate(petSite, result);
        if (!result.hasErrors()) {
            this.petSites.put(petSite.getName(), petSite.getUrl());
            status.setComplete();
            response.setRenderParameter("action", "list");
        }
    }
}
```

9.7 Almacenar atributos en una sesión con @SessionAttributes

La anotación @SessionAttributes se usa a nivel de tipo y declara los atributos de sesión utilizados por un controlador específico. Normalmente, esto mostrará una lista de los nombres de los atributos del modelo o los tipos atributos del modelo, que deben almacenarse de forma transparente en la sesión o en algún almacenamiento conversacional, utilizando los beans como respaldo entre las solicitudes posteriores.

El siguiente fragmento de código muestra el uso de esta anotación:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {  // ...}
```

9.8. Personalización WebDataBinder

Para personalizar parámetro de la petición con PropertyEditors, etc. a través de Spring WebDataBinder, se puede usar la anotación para métodos @InitBinder dentro del controlador o externalizar la configuración proporcionando un WebBindingInitializer personalizado.

Personalización del enlace de datos con @InitBinder

La anotación @InitBinder permite configurar los datos directamente dentro de la clase controlador. @InitBinder identifica los métodos que inicializan la WebDataBinder utilizada para rellenar los argumentos de objetos de formularios y de comando de los métodos de control de anotaciones.

Tales métodos soportan todos los argumentos que soporta @RequestMapping, excepto los objetos de comando/formulario y los correspondientes objetos de resultados de validación. Los métodos no deben tener un valor de retorno, por lo tanto, generalmente se declaran como void.

El siguiente ejemplo muestra el uso de @InitBinder para configurar un CustomDateEditor para todas las propiedades java.util.Date del formulario.

```
@Controller
public class MyFormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new
CustomDateEditor(dateFormat, false));
    }

    // ...

}
```

Configuración personalizada de un WebBindingInitializer

Para exteriorizar la inicialización del enlace de datos se puede proporcionar una implementación personalizada de la interfaz WebBindingInitializer, que luego se habilitara mediante el suministro de una configuración personalizada para el bean AnnotationMethodHandlerAdapter, anulando así la configuración por defecto.

10. Implementación de aplicaciones Portlets

El proceso de implementación de una aplicación Spring Portlets MVC no es diferente de la implementación de cualquier aplicación JSR-168 portlet. Sin embargo, esto es un poco confuso, por lo que vale la pena explicarlo brevemente.

En general, el contenedor portal/portlet se ejecuta en el contenedor de servlets de una aplicación y el portlets ejecuta en otro contenedor de servlets de otra aplicación web. Para que la aplicación web del contenedor de portlets pueda realizar llamadas a su portlet debe hacer llamadas a un servlet conocido que proporciona acceso a los servicios de los portlets definidos en su archivo portlet.xml.

La especificación JSR-168 no especifica exactamente cómo sucede esto, por lo que cada contenedor de portlet tiene su propio mecanismo, que por lo general implica algún tipo de "proceso de implementación" que hace los cambios en la propia aplicación web del portlet y luego registra los portlets dentro del contenedor de portlets.

Como mínimo, el archivo web.xml de la aplicación web del portlet se modifica para inyectar el servlet al que el contenedor de portlets llamará. En algunos casos, un solo servlet dará servicio a todos los portlets de la aplicación web, en otros casos habrá una instancia del servlet para cada portlet.

Algunos contenedores de portlets también inyectarán las bibliotecas y / o archivos de configuración en el webapp también. El contenedor de portlets también debe hacer su aplicación de la biblioteca de etiquetas JSP de portlet a disposición de su webapp.

En conclusión, es importante para entender las necesidades de implementación del portal de destino y asegurarse de que se cumplen (por lo general siguiendo el proceso de implementación automática que se proporciona).

PROGRAMANDO CON SPRING MVC

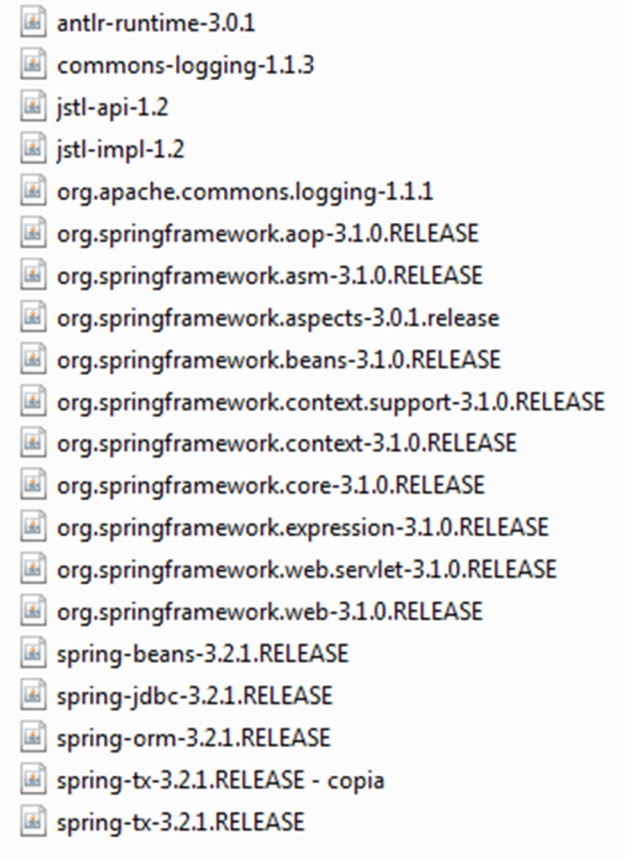
A lo largo de este trabajo se han visto los conceptos teóricos necesarios para realizar una aplicación web utilizando el framework Spring MVC, por eso en este capítulo se va a aplicar lo que se ha aprendido en los capítulos anteriores para programar una aplicación web, para ello se van a explicar las clases necesarias para hacerlo.

1. Entorno

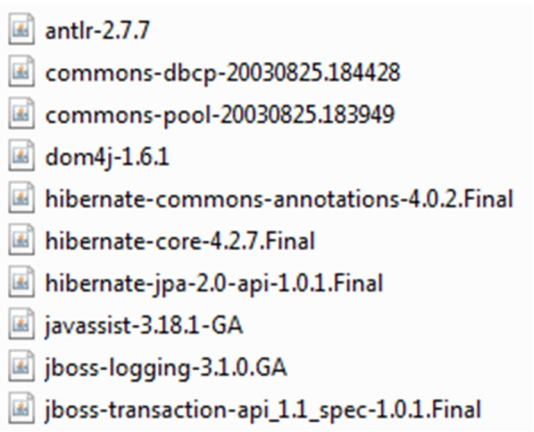
Para realizar este ejemplo se va a utilizar el entorno de desarrollo Eclipse en la versión Kepler, además utilizaremos el framework Hibernate para que se ocupe de la capa de persistencia de la aplicación, un servidor de base de datos MySQL y un servidor web para poder ejecutar la aplicación, en este caso utilizaremos el servidor Apache Tomcat 7.0.

En este ejemplo no vamos utilizar Maven para gestionar las dependencias por lo que tendremos que descargar las librerías de Spring e Hibernate y añadirlas a nuestro proyecto.

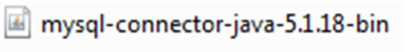
Las librerías de Spring necesarias son:



Las librerías de Hibernate que se van a necesitar son:



Además de estas librerías necesitaremos un conector de para la base de datos, en el caso de este ejemplo el conector para la base de datos MySQL.



2. La persistencia

En este apartado veremos las clases necesarias para realizar la persistencia de los datos.

HibernateUtil.java

```
package com.david.mayor.tfg.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory =
buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new
Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be
swallowed
            System.err.println("Initial SessionFactory creation
failed." +ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;    }}
```

Esta clase simplemente instanciará la SessionFactory que es la encargada de ejecutar las querys necesarias de la base de datos respectiva, si tuviéramos que usar más de una Base de datos tendríamos que definir otra SessionFactory pero en este caso solo necesitamos una.

UsuarioForm.java

```
package com.david.mayor.tfg.formulario;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="usuario")
public class UsuarioForm {
 @Id
 @Column(name="id")
 @GeneratedValue
 private Integer id;

 @Column(name="nombre")
 private String nombre;

 @Column(name="clave")
 private String clave;

 public Integer getId() {
  return id;
 }

 public void setId(Integer id) {
  this.id = id;
 }

 public String getNombre() {
  return nombre;
 }

 public void setNombre(String nombre) {
  this.nombre = nombre;
 }

 public String getClave() {
  return clave;
 }

 public void setClave(String clave) {
  this.clave = clave;
 }

 public String toString(){

  return "Usuario: "+this.nombre;
 }
}
```

Los import son las clases necesarias para mapear nuestra tabla USUARIO con la clase UsuarioForm. Hay que tener en cuenta que esto se hace para evitar crear un xml nuevo que se llame usuario.hbm.xml y mapearlo en ese xml. Esto se llama Hibernate con anotaciones los cuales facilitará la configuración del mapeo para tenerlo ya en nuestra clase respectiva y poder controlarla en el entorno del objeto.

La anotación @Entity indica que nuestra clase es una entidad que se encargará de mapear con una tabla de la base de datos, mientras que la anotación @Table(name="usuario") indica que la clase representará nuestra tabla USUARIO.

La anotación @Id, @Column(name="id"), @GeneratedValue da las características del atributo id y finalmente @Column(name="xxxxx") indica como estarán representadas los atributos con las respectivas columnas en la Tabla.

Con el mismo criterio se han realizado las clases **ComponenteForm.java** y **EquipoForm.java** por lo que no se añadirá nada más sobre ellas.

ComponenteDAO.java

```
package com.david.mayor.tfg.dao;

import java.util.List;

import com.david.mayor.tfg.formulario.ComponenteForm;

public interface ComponenteDAO {
  public void agregarComponente(ComponenteForm componente);
  public List<ComponenteForm> mostrarComponentes();
  public void eliminarComponente(Integer id);
  public void actualizarComponente(ComponenteForm componente);
  public ComponenteForm mostrarComponente(int id);
}
```

Esta clase es una interface que indica las acciones que se realizan en la tabla COMPONENTE a partir de su respectiva entidad mapeada, que en este caso es ComponenteForm. En consiguiente se implementará la interface modificando la siguiente clase: **ComponenteDAOImpl.java** tal como sigue:

```
package com.david.mayor.tfg.dao;

import java.util.ArrayList;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.david.mayor.tfg.formulario.ComponenteForm;
import com.david.mayor.tfg.util.HibernateUtil;


public class ComponenteDAOImpl implements ComponenteDAO{
 public void agregarComponente(ComponenteForm componente) {

        Transaction trns = null;
```

```
        Session session =
HibernateUtil.getSessionFactory().openSession();
        session.save(componente);

        try {
            trns = session.beginTransaction();
            session.save(componente);
            session.getTransaction().commit();
        } catch (RuntimeException e) {
            if (trns != null) {
                trns.rollback();
            }
            e.printStackTrace();
        } finally {
            session.flush();
            session.close();
        }
    }
 public List<ComponenteForm> mostrarComponentes() {
        List<ComponenteForm> componentes = new
ArrayList<ComponenteForm>();
        Transaction trns = null;
        Session session =
HibernateUtil.getSessionFactory().openSession();
        try {
            trns = session.beginTransaction();
            componentes = session.createQuery("from
ComponenteForm").list();
        } catch (RuntimeException e) {
            e.printStackTrace();
        } finally {
            session.flush();
            session.close();
        }

        return componentes;
 }
 public void eliminarComponente(Integer id) {
        Transaction trns = null;
        Session session =
HibernateUtil.getSessionFactory().openSession();
        try {
            trns = session.beginTransaction();
            ComponenteForm componente = (ComponenteForm)
session.load(ComponenteForm.class, new Integer(id));
            session.delete(componente);
            session.getTransaction().commit();
        } catch (RuntimeException e) {
            if (trns != null) {
                trns.rollback();
            }
            e.printStackTrace();
        } finally {
            session.flush();
            session.close();
        }
 }

    public void actualizarComponente(ComponenteForm componente) {
        Transaction trns = null;
```

```
        Session session =
HibernateUtil.getSessionFactory().openSession();
        try {
            trns = session.beginTransaction();
            session.update(componente);
            session.getTransaction().commit();
        } catch (RuntimeException e) {
            if (trns != null) {
                trns.rollback();
            }
            e.printStackTrace();
        } finally {
            session.flush();
            session.close();
        }
    }

    public ComponenteForm mostrarComponente(int id) {
     ComponenteForm componente = null;
        Transaction trns = null;
        Session session =
HibernateUtil.getSessionFactory().openSession();
        try {
            trns = session.beginTransaction();
            String queryString = "from ComponenteForm where id = :id";
            Query query = session.createQuery(queryString);
            query.setInteger("id", id);
            componente = (ComponenteForm) query.uniqueResult();
        } catch (RuntimeException e) {
            e.printStackTrace();
        } finally {
            session.flush();
            session.close();
        }
        return componente;
    }
}
```

En esta implementación básicamente desarrollamos el CRUD de la tabla COMPONENTE. Lo más importante aquí es tener claro que es necesario abrir y cerrar la sesión (el uso exclusivo de la tabla) antes de cualquier acción: openSession() y session.close() respectivamente.

Posteriormente las acciones de agregar, actualizar, eliminar están definidas en los métodos del objeto session de la siguiente forma respectivamente: session.save(componente); session.update(componente); session.delete(componente);

Hay que tener en cuenta que las querys no tienen que realizarse contra las tablas de la Base de datos si no contra la entidad que la representa, en este caso la tabla COMPONENTE está representada por la entidad ComponenteForm. Por ello las querys están dadas de la siguiente forma: "from ComponenteForm" o "from ComponenteForm where id = :id", los atributos también serán usados con el mismo criterio es decir las querys se realizarán con los atributos de la entidad que representan las filas de la tabla.

Finalmente para obtener un fila, o todas, también se usa un método del objeto session que es session.createQuery(queryString), donde queryString viene a ser la cadena con los datos de la

restricción para la búsqueda ya sea de uno o más elementos. Cuando se va a obtener un solo elemento es necesario especificar query.uniqueResult().

Con el mismo criterio se han realizado las clases **EquipoDAO.java** y **EquipoDAOImpl.java** por lo que no se añadirá nada más sobre ellas.

Por otro lado la clase **UsuarioDAO.java** sigue los mismos criterios que sus homologas, pero no es así con **UsuarioDAOImpl.java**

```
package com.david.mayor.tfg.dao;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;



import com.david.mayor.tfg.formulario.UsuarioForm;
import com.david.mayor.tfg.util.HibernateUtil;

public class UsuarioDAOImpl implements UsuarioDAO{

    public boolean buscarUsuario(UsuarioForm usuario) {

     boolean existe=false;
        Transaction trns = null;
        Session session =
HibernateUtil.getSessionFactory().openSession();
        try {
            trns = session.beginTransaction();
            String queryString = "from UsuarioForm where nombre =
:nombre and clave = :clave";
            Query query = session.createQuery(queryString);
            query.setString("nombre", usuario.getNombre());
            query.setString("clave", usuario.getClave());
            existe = !(((UsuarioForm) query.uniqueResult())==(null));
            //Aqui verificamos si la query trae algun resultado, si no
es asi el usuario no existe.
        } catch (RuntimeException e) {
         existe = false;
            e.printStackTrace();
        } finally {
            session.flush();
            session.close();
        }
        return existe;
    }
}
```

Aquí no se utiliza ningún CRUD porque solo será necesario validar que el usuario exista para poderlo dejar entrar a nuestro sistema por eso se valida con la query "from UsuarioForm where nombre = :nombre and clave = :clave", recordar la consulta se realiza sobre la entidad UsuarioForm y no sobre la tabla USUARIO.

3. Capa de Servicios

En este apartado veremos las clases necesarias para realizar la capa de servicios de la aplicación web, para ello es necesario crear una interfaz y su correspondiente implementación cuya única función será la de ser instanciada e invocada por el controlador y a su vez invocar los métodos del DAO que sean necesarios. De manera similar se realizan las interfaces y las implementaciones para cada DAO por lo que solo se mostraran como ejemplo **EquipoService.java** y **EquipoServiceImpl.java.**

EquipoService.java

```
package com.david.mayor.tfg.servicios;

import java.util.List;

import com.david.mayor.tfg.formulario.EquipoForm;

public interface EquipoService {
        public void agregarEquipo(EquipoForm equipo);
        public List<EquipoForm> mostrarEquipos();
        public void eliminarEquipo(Integer id);
        public void actualizarEquipo(EquipoForm equipo);
        public EquipoForm mostrarEquipo(int id);
}
```

EquipoServiceImpl.java

```
package com.david.mayor.tfg.servicios;

import java.util.List;

import com.david.mayor.tfg.dao.EquipoDAO;
import com.david.mayor.tfg.dao.EquipoDAOImpl;
import com.david.mayor.tfg.formulario.EquipoForm;



public class EquipoServiceImpl implements EquipoService{

         private EquipoDAO equipoDAO = new EquipoDAOImpl();

         public void agregarEquipo(EquipoForm equipo) {
          // TODO Auto-generated method stub
                 equipoDAO.agregarEquipo(equipo);
         }

         public List<EquipoForm> mostrarEquipos() {
          // TODO Auto-generated method stub
          return equipoDAO.mostrarEquipos();
         }

         public void eliminarEquipo(Integer id) {
          // TODO Auto-generated method stub
          equipoDAO.eliminarEquipo(id);
         }
```

```
        public void actualizarEquipo(EquipoForm equipo) {
         // TODO Auto-generated method stub
         equipoDAO.actualizarEquipo(equipo);
        }

        public EquipoForm mostrarEquipo(int id) {
         // TODO Auto-generated method stub
         return equipoDAO.mostrarEquipo(id);
        }
}
```

4. Los Controladores

A continuación se mostraran los controladores que implementara la aplicación.

UsuarioController.java

```
package com.david.mayor.tfg.controlador;

import com.david.mayor.tfg.formulario.ComponenteForm;
import com.david.mayor.tfg.formulario.UsuarioForm;
import com.david.mayor.tfg.servicios.UsuarioService;
import com.david.mayor.tfg.servicios.UsuarioServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;


@Controller
public class UsuarioController {
 private UsuarioService usuarioService = new UsuarioServiceImpl();
 static UsuarioForm usuarioForm = new UsuarioForm();


    @RequestMapping(value = "/inicializarLogin.html", method =
RequestMethod.GET)
    public ModelAndView inicializarUsuario() {
     return new ModelAndView("login" , "usuarioForm", new
UsuarioForm());
    }

    @RequestMapping(value = "/verificarLogin.html", method =
RequestMethod.POST)
    public ModelAndView
verificarUsuario(@ModelAttribute("usuarioForm") UsuarioForm
usuarioForm) {
        boolean existe = false;
     existe = usuarioService.buscarUsuario(usuarioForm);


if("".equals(usuarioForm.getNombre())&&"".equals(usuarioForm.getClave(
))){
```

```
      return new ModelAndView("login" , "mensaje", "Debe de llenar los
campos de Usuario y Clave");
     }
     else if(existe){

UsuarioController.usuarioForm.setNombre(usuarioForm.getNombre());
      UsuarioController.usuarioForm.setClave(usuarioForm.getClave());
      ModelAndView modelo= new ModelAndView("menuPrincipal" ,
"mensaje", "Usuario Correcto");
      modelo.addObject("mensajeMenu","Menu Principal");
      modelo.addObject("usuarioForm", usuarioForm);
      return modelo;
     }
     else{
      return new ModelAndView("login" , "mensaje", "Usuario
Incorrecto");
     }
    }

    @RequestMapping(value = "/volverMenu.html", method =
RequestMethod.GET)
    public ModelAndView volverMenu() {
        ModelAndView modelo= new ModelAndView("menuPrincipal" ,
"mensaje", "Usuario");
        modelo.addObject("usuarioForm", UsuarioController.usuarioForm);
        return modelo;
    }
}
```

Todos los import son importantes puesto que son las librerías de Spring que necesitamos para desarrollarlo. La anotación @Controller es esencial para indicar que la clase UsuarioController es un controlador.

Se define el atributo UsuarioForm usuarioForm de tipo estático para que guarde el usuario y clave a validar en el login.Definimos el @RequestMapping para indicar que será el método que se ejecutará cuando se invoque desde una página con el nombre de inicializarLogin.html. Inmediatamente después se define el method = RequestMethod.GET ya que para este caso no estaremos pasando ninguna información.

Una vez definido el tipo de método se definirá el método en sí, donde solo se devolverá el objeto UsuarioForm usuarioForm vacío definiéndolo de la siguiente manera: return new ModelAndView("login" , "usuarioForm", new UsuarioForm())

Luego se define otro @RequestMapping donde el method será del tipo method = RequestMethod.POST ya que en este caso si necesitaremos información de la página que la invoca, esta información serán los datos del usuario que se han ingresado.

Luego se utilizará la capa de servicio del usuario: UsuarioService para verificar que el usuario tiene los datos correctos y existe con el nombre y clave ingresado.

Una vez se haya validado el usuario se realiza la verificación principal que es el "else if(existe)" ya que si el usuario y clave coinciden se crea un objeto del tipo ModelAndView que llevará a la página del menú principal con el mensaje correcto y con la información del usuario que se

logueó mediante modelo.addObject("usuarioForm", usuarioForm), en caso contrario si la validación no resultó exitosa se vuelve a la página login.jsp con su mensaje respectivo.

ComponenteController.java

```
package com.david.mayor.tfg.controlador;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import com.david.mayor.tfg.formulario.ComponenteForm;
import com.david.mayor.tfg.servicios.ComponenteService;
import com.david.mayor.tfg.servicios.ComponenteServiceImpl;

@Controller
public class ComponenteController {

 private ComponenteService componenteService= new
ComponenteServiceImpl();

 @RequestMapping(value="/agregarComponentes.html", method =
RequestMethod.POST)
 public ModelAndView
guardarComponente(@ModelAttribute("componenteForm") ComponenteForm
componente) {
  componenteService.agregarComponente(componente);
  return new ModelAndView("mostrarComponentes" , "listaComponentes",
componenteService.mostrarComponentes());
 }

 @RequestMapping(value="/actualizarComponentes.html", method =
RequestMethod.POST)
 public ModelAndView
actualizarComponente(@ModelAttribute("componenteForm") ComponenteForm
componente) {
  componenteService.actualizarComponente(componente);
  return new ModelAndView("mostrarComponentes" , "listaComponentes",
componenteService.mostrarComponentes());
 }

 @RequestMapping(value="/eliminarComponentes.html/{componenteId}",
method = RequestMethod.GET)
 public ModelAndView eliminarComponente(@PathVariable("componenteId")
Integer componenteId) {
  componenteService.eliminarComponente(componenteId);
  return new ModelAndView("mostrarComponentes" , "listaComponentes",
componenteService.mostrarComponentes());
 }

 @RequestMapping(value="/modificarComponentes.html/{componenteId}",
method = RequestMethod.GET)
 public ModelAndView modificarComponente(@PathVariable("componenteId")
Integer componenteId) {
  ComponenteForm
componente=componenteService.mostrarComponente(componenteId);
  ModelAndView modelo= new ModelAndView("agregarComponentes" ,
"mensaje", "Usuario Correcto");
```

```
  modelo.addObject("mensajeComponente","Modificar Componente");
  modelo.addObject("usuarioForm", UsuarioController.usuarioForm);
  modelo.addObject("componenteForm", componente);
  return modelo;
 }

 @RequestMapping(value="/pantallaComponente.html", method =
RequestMethod.GET)
 public ModelAndView volverComponente() {
  ModelAndView modelo= new ModelAndView("agregarComponentes" ,
"mensaje", "Usuario Correcto");
  modelo.addObject("mensajeComponente","Agregar Componente");
  modelo.addObject("usuarioForm", UsuarioController.usuarioForm);
  modelo.addObject("componenteForm", new ComponenteForm());
  return modelo;
 }

 @RequestMapping(value="/mostrarComponentes.html", method =
RequestMethod.GET)
 public ModelAndView mostrarComponentes() {
  return new ModelAndView("mostrarComponentes" , "listaComponentes",
componenteService.mostrarComponentes());
 }
}
```

El método guardarComponente que es de tipo POST puesto que recibe el formulario con la información del registro que deseamos guardar en la tabla COMPONENTE, invoca al ComponenteService para realizar la tarea de guardado respectiva componenteService.agregarComponente(componente). Una vez guardado el dato se le pasará a la vista una lista de todos los registros que tenemos en la tabla COMPONENTE: componenteService.mostrarComponentes().

El método actualizarComponente que es de tipo POST puesto que recibe el formulario con la información del registro que deseamos actualizar en la tabla COMPONENTE, invoca al ComponenteService para realizar la tarea de actualización respectiva componenteService.actualizarComponente(componente). Una vez actualizado el dato se le pasará a la vista una lista de todos los registros que tenemos en la tabla COMPONENTE: componenteService.mostrarComponentes().

El método eliminarComponente que es de tipo GET puesto que no recibe ningún formulario, sin embargo se necesita el id del componente que se desea eliminar por lo que recibe una variable del tipo @PathVariable("componenteId") que será pasada en el RequestMapping, con este valor se invoca al ComponenteService para realizar la tarea de eliminación respectiva componenteService.eliminarComponente(componenteId). Una vez eliminado el componente se le pasará a la vista una lista de todos los registros que tenemos en la tabla COMPONENTE: componenteService.mostrarComponentes().

El método modificarComponente que es de tipo GET puesto que no recibe ningún formulario, sin embargo tal como en el caso anterior se necesita el id del componente que deseamos actualizar por lo que se recibe una variable del tipo @PathVariable("componenteId") que será pasado en el RequestMapping, con este valor se invoca al ComponenteService para obtener los datos del componente a actualizar

componenteService.mostrarComponente(componenteId). Una vez conseguidos los datos del componente a modificar se le pasará a la vista este ComponenteForm para que se pueda mostrar en un formulario modelo.addObject("componenteForm", componente).

El método volverComponente que es de tipo GET puesto que no recibe ningún formulario, no se comunica con un servicio por que su objetivo es volver a la vista de ingresar un componente con todos los datos del formulario vacío modelo.addObject("componenteForm", new ComponenteForm()).

El método mostrarComponentes que es de tipo GET puesto que no recibe ningún formulario simplemente le pasa a la vista la lista de todos componentes que existen en la tabla sin realizar ninguna otra acción componenteService.mostrarComponentes().

El controlador **Equipo.Controller.java** es similar a este último controlador.

5. Las vistas

En este apartado se explicaran las etiquetas utilizadas en las vistas para el correcto funcionamiento de la aplicación.

Index.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<jsp:forward page="inicializarLogin.html"></jsp:forward><span
style="font-family: Times New Roman;"><span style="white-space:
normal;">
</span></span>
```

Que básicamente redirecciona index.jsp a login.jsp

login.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-
1">
<title>Proyecto Spring TFG</title>
</head>
<body>
<c:if test="${mensaje !='Usuario Correcto'}">
  <h4>${mensaje}</h4>
</c:if>
```

```
<spring:url var="verificarLogin" value="/verificarLogin.html"/>

<form:form id="login" modelAttribute="usuarioForm" method="post"
action="${verificarLogin}">
 <table width="400px" height="150px">
  <tr>
  <td><form:label path="nombre">Nombre</form:label></td>
  <td><form:input  path="nombre"/></td>
  </tr>
  <tr>
  <td><form:label path="clave">Clave</form:label></td>
  <td><form:input  path="clave"/></td>
  </tr>
  <tr><td></td><td>
  <input type="submit" value="Login" />
  </td></tr>
 </table>
</form:form>
</body>
</html>
```

Aquí se define primero que todo el mensaje que se mostrará en caso de algún error del login aunque inicialmente ese mensaje estará vacío, en caso contrario mostrará el mensaje respectivo. Luego se define el @RequestMapping verificarLogin.html al cual invocará nuestro jsp cuando se realice el submit mediante el login a traves del atributo action.

Luego se define el form donde nuestro objeto será usuarioForm de method post ya que enviaremos información al servlet y la acción a realizar con el submit action="${verificarLogin} que definimos en var="verificarLogin" value="/verificarLogin.html". Finalmente se insertan los input necesarios que son el nombre y la clave, ojo que estos son atributos de la clase UsuarioForm tal como lo definimos anteriormente.

Hay que ponerle un énfasis especial a la forma como se define la URL: <spring:url var="verificarLogin" value="/verificarLogin.html"/> se usa de esta forma y no <c:url var="verificarLogin" value="verificarLogin.html"/> porque de esta última forma corremos el peligro de engañar al servlet con otro contextpath. Al usar spring:url ya estamos dándole el contextpath correcto a nuestras llamadas ya sean por GET o POST.

menuPrincipal.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-
1">
<title>Menu Principal</title>
</head>
<body>
```

```
 <h4>${mensaje} : ${usuarioForm.nombre}</h4>
 <spring:url var="salir" value="/inicializarLogin.html"/>
 <a href="${salir}" >Salir</a>
<br>
<br>
 <spring:url var="componente" value="/pantallaComponente.html"/>
<a href="${componente}" >Agrega un nuevo componente</a>
<br>
<spring:url var="mcomponente" value="/mostrarComponentes.html"/>
<a href="${mcomponente}" >Ver Lista de componentes</a>
<br>
<spring:url var="equipo" value="/pantallaEquipo.html"/>
<a href="${equipo}" >Agrega un nuevo equipo</a>
<br>
<spring:url var="mequipo" value="/mostrarEquipos.html"/>
<a href="${mequipo}" >Ver Lista de equipos</a>
</body>
</html>
```

En esta vista lo que se hace es mostrar las distintas opciones para movernos por las páginas de la aplicación mediante las opciones establecidas en los controladores.

agregarComponentes.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Componentes</title>
</head>
<body>
 <spring:url var="volver" value="/volverMenu.html"/>
 <a href="${volver}" >Volver al menú</a>

<h4>${mensajeComponente}</h4>
<c:choose>
    <c:when test="${mensajeComponente !='Agregar Componente'}">
        <!--<c:url var="accionComponente"
value="actualizarComponentes.html"/>-->
        <spring:url var="accionComponente"
value="/actualizarComponentes.html"/>
    </c:when>
    <c:otherwise>
        <!--<c:url var="accionComponente" value="agregarComponentes.html"/>--
>
        <spring:url var="accionComponente" value="/agregarComponentes.html"/>
    </c:otherwise>
</c:choose>

<form:form id="agregar" modelAttribute="componenteForm" method="post"
action="${accionComponente}">
 <table width="400px" height="150px">
  <tr>
```

```
  <form:hidden path="id" />
  <td><form:label path="nombre">Nombre</form:label></td>
  <td><form:input  path="nombre"/></td>
  </tr>
  <tr>
  <td><form:label path="version">Versión</form:label></td>
  <td><form:input  path="version"/></td>
  </tr>
  <tr>
  <td><form:label path="tipo">Tipo</form:label></td>
  <td><form:input  path="tipo"/></td>
  </tr>

  <tr><td></td><td>
  <c:choose>
      <c:when test="${mensajeComponente !='Agregar Componente'}">
          <input type="submit" value="Modificar" />
      </c:when>
      <c:otherwise>
          <input type="submit" value="Agregar" />
      </c:otherwise>
  </c:choose>
  </td></tr>
 </table>
</form:form>
<spring:url var="lista" value="/mostrarComponentes.html"/>
<a href="${lista}" >Ver Lista de componentes</a>
</body>
</html>
```

Aquí se verifica que de acción venimos si es para agregar un componente o para modificarlo y así mostrar el mensaje y acciones respectivas. Estas acciones están definidas en el controlador. Por otro lado se toman los datos de un componente a través del formulario componenteForm y en el submit los dirigimos a @RequestMapping agregarComponentes.html ya declarado en el ComponenteController como método POST.

mostrarComponentes.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
    <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
    <%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Lista de Componentes</title>
</head>
<body>
 <spring:url var="volver" value="/volverMenu.html"/>
 <a href="${volver}" >Volver al menú</a>
<center>
<br><br><br><br><br><br>
<div style="color: teal;font-size: 30px">Lista de Componentes</div>
<br><br>
<c:if test="${!empty listaComponentes}">
<table border="1" bgcolor="black" width="600px">
```

```
<tr style="background-color: teal;color: white;text-align: center;"
height="40px">
<td>Nombre</td>
<td>Version</td>
<td>Tipo</td>
<td> </td>
<td> </td>
</tr>
<c:forEach items="${listaComponentes}" var="componente">
<tr style="background-color:white;color: black;text-align: center;"
height="30px" >
<td><c:out value="${componente.nombre}"/></td>
<td><c:out value="${componente.version}"/></td>
<td><c:out value="${componente.tipo}"/></td>
<spring:url var="modificar"
value="/modificarComponentes.html/${componente.id}"/>
<spring:url var="eliminar"
value="/eliminarComponentes.html/${componente.id}"/>
 <td><a href="${modificar}" class="parent">Modificar</a></td>
  <td><a href="${eliminar}" class="parent">Eliminar</a></td>
</tr>
</c:forEach>
</table>
</c:if>
<br>
<spring:url var="volver" value="/pantallaComponente.html"/>
<a href="${volver}" >Agrega un nuevo componente</a>
</center>
</body>
</html>
```

Aquí se recorre la lista de los componentes que hay en la base de datos y que ha pasado el controlador listaComponentes siempre y cuando listaComponentes no sea nula {!empty listaComponentes}.Luego por cada registro que se obtenga y se pinte se le añadirán dos links para que se puedan modificar o eliminar respectivamente {modificar}, {eliminar}.

Hay que poner especial atención en como pasaremos la información del id del objeto que se está seleccionando: value="/modificarComponentes.html/${componente.id}", como se ve se está adjuntando el id en el RequestMapping que tomará el controlador posteriormente. Finalmente generamos un link para volver a agregar otro componente: volverComponente.html

Las vistas agregarEquipo.jsp y mostrarEquipo.jsp son similares a estas dos últimas vistas.

6. La configuración

Para terminar este capítulo se mostraran explicaran los distintos archivos de configuración necesarios para el correcto funcionamiento de Hibernate y de Spring.

Hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
 <session-factory>
        <!-- Database connection settings -->
        <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property
name="connection.url">jdbc:mysql://localhost/BD_SPRINGMVC</property>
        <property name="connection.username">admin</property>
        <property name="connection.password">test</property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property
name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -
->
        <property
name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache -->
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</prope
rty>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">update</property>
        <mapping class="com.david.mayor.tfg.formulario.UsuarioForm" />
        <mapping class="com.david.mayor.tfg.formulario.EquipoForm" />
        <mapping class="com.david.mayor.tfg.formulario.ComponenteForm"
/>
 </session-factory>
</hibernate-configuration>
```

En esta configuración lo que estamos definiendo principalmente es el driver (conector) que usaremos: com.mysql.jdbc.Driver, la base de datos: jdbc:mysql://localhost/BD_SPRINGMVC, el usuario y la contraseña para acceder a la base de datos, el dialecto que usaremos que en este caso es del MySQL: org.hibernate.dialect.MySQLDialect y las clases entidades que representarán las tablas de la base de datos que definimos al inicio del capítulo: UsuarioForm, EquipoForm y ComponenteForm.

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <display-name>ProyectoSpringTFG</display-name>
    <welcome-file-list>
   <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
    <servlet>
        <servlet-name>proyectoSpring</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>proyectoSpring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Donde hay que percatarse que el nombre del servlet proyectoSpring tendrá que ser definido posteriormente como proyectoSpring-servlet.xml (agregando el sufijo -servlet.xml) para que esté correctamente configurado.

Aquí hay que tener en cuenta que el url-pattern se ha puesto como / , porque si se pone como *.html o como *.jsp no funcionará para este proyecto puesto que al pasar datos al controlador del modo: /modificarComponentes.html/${componente.id} la url no termina en html ni en jsp por lo que saldría un error de url es por ello que al definir este pattern / ya se acepta cualquier tipo de url.

proyectoSpring-servlet.xml

```
<?xml  version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:annotation-config />
    <context:component-scan base-
package="com.david.mayor.tfg.controlador" />

    <bean id="jspViewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolv
er">
        <property name="viewClass"
```

```
            value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

Donde se define el paquete donde se buscarán los controladores, el tipo de Resolver InternalResourceViewResolver y se pone el prefijo y sufijo de las paginas a desplegar.

CONCLUSIONES

Después de dar por finalizado este Proyecto Fin de Carrera "Evaluación Spring MVC", hacemos balance sobre los conocimientos adquiridos y las impresiones obtenidas a lo largo del mismo.

La utilización de un framework, y en especial Spring MVC, puede proporcionar mejoras y beneficios a la hora de desarrollar aplicaciones, ya que brinda facilidades a la de escribir código, diseñar la aplicación, etc. La más destacable es que proporciona una estructura más sólida y consistente, que hace que cualquier persona que posea los conocimientos necesarios pueda entender con facilidad la aplicación. Aunque hay que destacar que la adquisición de estos conocimientos no se consiguen con facilidad ya que es necesario una gran cantidad de tiempo para comprender y poder utilizar el framewrok correctamente.

Otra cuestión que se observó fue que Spring es un framework que está formado por diferentes módulos, lo que facilita su integración con varias herramientas enfocadas cada una en un área particular, como puede ser Hibernate o JPA para acceder a la base de datos, utilizar otros framework cono Struts o utilizar una multitud de tecnologías para representar las vistas como pueden ser JSP, Tiles, PDF, Excel y muchas más, por lo que se puede decir que Spring es un framework diseñado para integrarse con otras tecnologías y no de competir con ellas.

Para finalizar hay que destacar que el patrón de diseño MVC brinda un buen número de funcionalidades a las aplicaciones web, en especial la reutilización de código, así como un mejor diseño y sobre todo modularidad.

BIBLIOGRAFÍA

[1] http://www.oracle.com/technetwork/java/mvc-140477.html

[2] http://spring.io/

[3] Spring Recipres. Gary Mack, Josh Long y Daniel Rubio 2010.

[4] Spring Web Flow 2 Web Development. Sven Lüppken, Markus Stäuble 2009.

[5] Spring. Craig Walls. Anaya Multimedia 2011.